致尊敬的顾客

关于产品目录等资料中的旧公司名称

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

瑞萨电子公司网址：http://www.renesas.com

2010年4月1日
瑞萨电子公司

【发行】瑞萨电子公司（http://www.renesas.com）

【业务咨询】http://www.renesas.com/inquiry

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

User's Manual

M16C/60、M16C/30、M16C/20、M16C/10、M16C/Tiny、 R8C/Tiny 系列 C 编译器套件 V.5.43-C 编译器用户手册

瑞萨电子
www.renesas.com
Rev.1.00 2007.09

- Microsoft、MS-DOS、Windows 和 Windows NT 是 Microsoft Corporation 在美国及其它国家/地区的注册商标或商标。HP-UX 是 Hewlett-Packard Company 的注册商标。
- Sun、 Java 和所有 Java 相关的商标和标志是 Sun Microsystems, Inc. 在美国或其它国家/地区的商标或注册商标，并在经许可的情况下使用。
- UNIX 是 The Open Group 在美国和其它国家/地区的注册商标。
- Linux 是 Linus Torvalds 的商标。
- Turbolinux 及其商标是 Turbolinux, Inc. 的商标。
- IBM 和 AT 是 International Business Machines Corporation 的注册商标。
- HP9000 是 Hewlett-Packard Company 的产品名称。
- SPARC 和 SPARCstation 是 SPARC International, Inc. 的注册商标。
- Intel 和 Pentium 是 Intel Corporation 的注册商标。
- Adobe 和 Acrobat 是 Adobe Systems Incorporated 的注册商标。
- Netscape 和 Netscape Navigator 是 Netscape Communications Corporation 在美国和其它国家/地区的注册商标。

所有其它品牌和产品名称是它们各自所有者的商标、注册商标或服务标志。

请遵循安全第一进行电路设计

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

关于利用本资料时的注意事项

- 本资料是为了让用户根据用途选择合适的瑞萨科技产品的参考资料，不转让属于瑞萨科技或者第三方所有的知识产权和其它权利的许可。
- 对于因使用本资料所记载的产品数据、图、表、程序、算法以及其它应用电路的例子而引起的损害或者对第三者的权力的侵犯，瑞萨科技和瑞萨解决方案不承担责任。
- 本资料所记载的产品数据、图、表、程序、算法以及其它所有信息均为本资料发行时的信息，由于改进产品或者其它原因，本资料记载的信息可能变动，恕不另行通知。在购买本资料所记载的产品时，请预先向瑞萨科技、瑞萨解决方案或者经授权的瑞萨科技产品经销商确认最新信息。本资料所记载的信息可能存在技术不准确或者印刷错误。因这些错误而引起的损害、责任问题或者其它损失，瑞萨科技和瑞萨解决方案不承担责任。同时也请注意瑞萨科技和瑞萨解决方案通过各种方式公布的信息，包括瑞萨网站 (http：//www.renesas.com)。
- 在使用本资料所记载部分或者全部数据、图、表、程序以及算法等信息时，在最终做出有关信息和产品是否适用的判断前，务必对作为整个系统的所有信息进行评价。由于本资料所记载的信息而引起的损害、责任问题或者其它损失，瑞萨科技和瑞萨解决方案不承担责任。
- 瑞萨科技的半导体产品不是为在可能和人命相关的环境下使用的设备或者系统而设计和制造的产品。在研讨将本资料所记载的产品用于运输、交通车辆、医疗、航天、原子能控制、海底中继器的设备或者系统等特殊用途时，请与瑞萨科技、瑞萨解决方案或者经授权的瑞萨科技产品经销商联系。
- 未经瑞萨科技和瑞萨解决方案的书面许可，不得翻印或者复制全部或者部分资料的内容。
- 如果本资料所记载的某产品或者技术内容受日本出口管理限制，必须在得到日本政府的有关部门许可后才能出口，并且不准进口到批准目的地国家以外的国家。禁止违反日本和 （或者）目的地国家的出口管理法和法规的任何转卖、挪用或者再出口。
- 如果需要了解本资料所记载的信息或产品详情，请与瑞萨科技或瑞萨解决方案联系。

有关对本资料内容或产品的查询，请填妥安装程序在下列目录中生成的文本文件，并将它电邮给您当地的经销商。

\SUPPORT\Product-name\SUPPORT.TXT

瑞萨工具网站 http://www.renesas.com/en/tools

前言

NC30 是为瑞萨 M16C/60、M16C/30、M16C/Tiny、M16C/20、M16C/10、R8C/Tiny 系列提供的 C 编译系统。NC30 为 M16C/60、M16C/30、M16C/Tiny、M16C/20、M16C/10、R8C/Tiny 系列将用 C 编写的程序转换为汇编语言源文件。您也可以通过指定编译器选项，汇编和连接，以生成可以写入到单片机的十六进制文件。

在使用 NC30 前，请先阅读本手册中的注意事项。

术语说明

本手册中使用的术语列出如下。

术语	含义
NC30	本套件中的编译系统
nc30	编译驱动器及其可执行文件
AS30	在本编译器中的汇编器套件
as30	可再定位的宏的汇编器及其可执行文件

符号说明

本手册中使用的符号如下。

符号	说明
#	Root 用户提示符
%	UNIX 提示符
A>	MS-Windows(TM) 提示符
<RET>	回车键
< >	强制性项目
[ ]	可选项目
Δ	空格或制表符代码 （强制性）
▲	空格或制表符代码 （可选）
: （已省略） :	表示该文件列表部分已省略

对其它符号的说明将在使用时提供。

目　录

1. NC30 简介

本章介绍 NC30 所执行的编译处理，并提供一个使用 NC30 的程序开发范例。

1.1 NC30 的组件

NC30 由下列八个可执行文件组成：

1. nc30 ..... 编译驱动器
2. cpp30 ..... 预处理器
3. ccom30 ..... 编译器
4. aopt30 ..... 汇编优化器
5. sbauto ..... SB 寄存器自动更换工具
6. StkViewer & stk ..... STK 查看器与堆栈大小计算工具
7. utl30 ..... SBDATA 声明及 SPECIAL 页函数声明工具
8. MapViewer ..... 映射查看器

1.2 NC30 处理流程

图 1.1 说明 NC30 的处理流程。

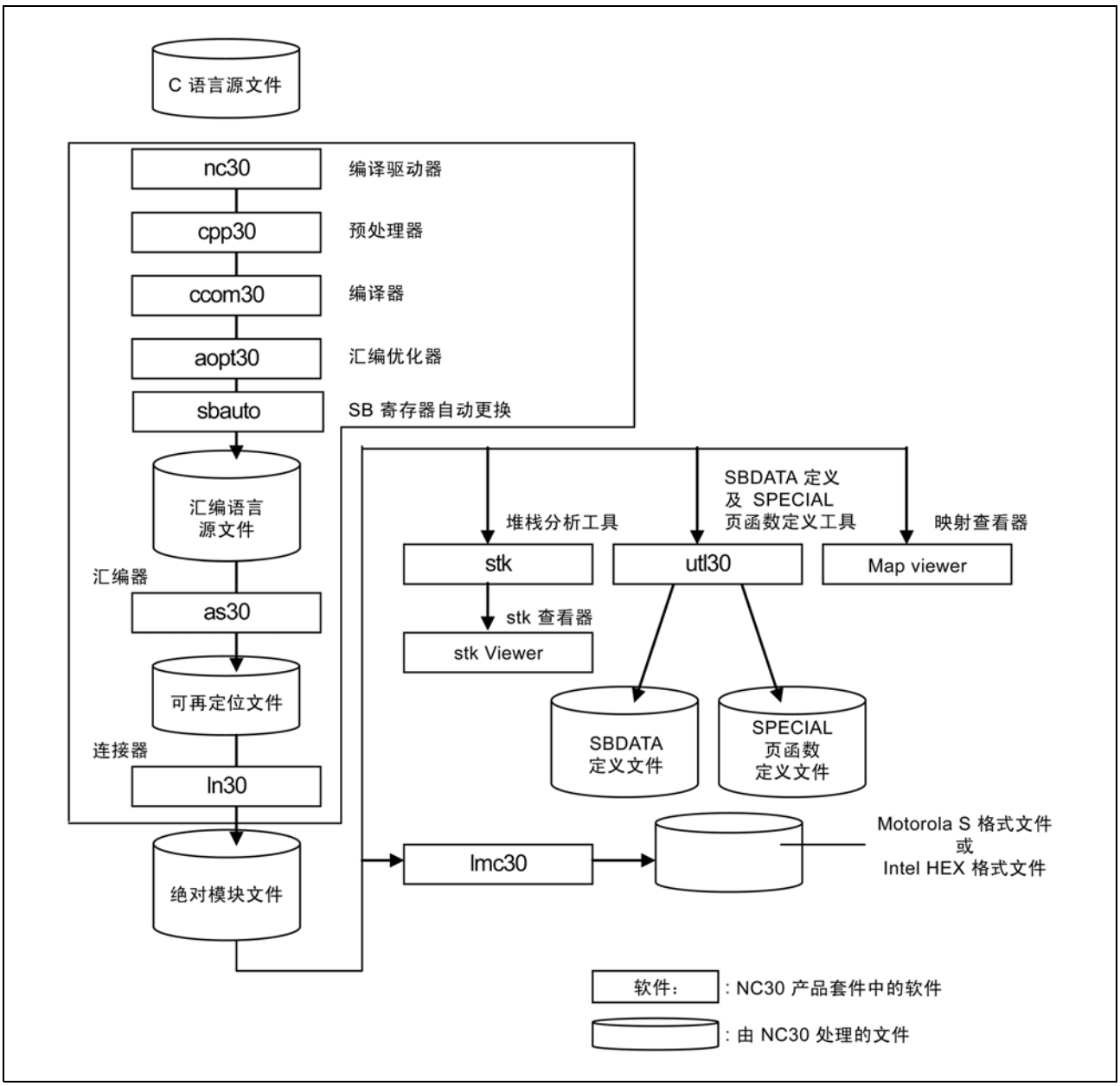


图 1.1　　NC30 处理流程

1.2.1 nc30

nc30 是编译驱动器的可执行文件。

通过指定选项， nc30 可执行从编译到连接的一系列操作。通过在启动 nc30 时指定 -as30 和 -ln 命令行选项，您也可以指定 as30 可再定位的宏的汇编器及 ln30 连接编辑器。

1.2.2 cpp30

cpp30 是预处理器的可执行文件。

cpp30 处理以 # （#define、 #include 等）开头的宏，并执行条件编译 （#if-#else-#endif 等）。

1.2.3 ccom30

ccom30 是编译器本身的可执行文件。

由 cpp30 处理的 C 源程序，被转换为可被 as30 处理的汇编语言源程序。

1.2.4 aopt30

aopt30 是汇编优化器。

它优化 ccom30 所输出的汇编器代码。

1.2.5 sbauto

sbauto 根据编译器所输出的监视信息，来分析一个函数中的外部变量被引用的次数，并输出最优 SB 相对值。

1.2.6 StkViewer & stk

StkViewer 是以图形方式显示程序操作所需的堆栈大小，及函数调用关系的工具的执行文件。另外， stk 则是分析 StkViewer 所需信息的工具的执行文件。

StkViewer 会调用 stk 以处理添加到绝对模块文件 (.x30) 的监视[1] 信息，查找程序操作所需的堆栈大小和函数调用关系，并显示结果。

另外，若有无法完全用监视信息来分析的信息，通过指定此信息， StkViewer 将重新计算堆栈大小和函数调用关系，并显示结果。

若要使用 StkViewer & stk，在编译时指定编译驱动器启动选项 -finfo，以便将监视信息添加到绝对模块文件 (.x30)。

1.2.7 utl30

utl30 是 SBDATA 声明工具及 SPECIAL 页函数声明工具的执行文件。

通过处理绝对模块文件 (.x30)， utl30 将生成一个包含 SBDATA 声明 （位于 SB 区内，从最常用的一个开始）的文件，及一个包含 SPECIAL 页函数声明 （位于 SPECIAL 页区内，从最常用的一个开始）的文件。

若要使用 utl30，在编译时指定编译驱动器启动选项 -finfo，以便生成绝对模块文件 (.x30)。

1.2.8 MapViewer

MapViewer 是映射查看器的执行文件。

通过处理绝对模块文件 (.x30)， MapViewer 将以图形方式显示连接后的存储器映射。

若要使用 MapViewer，在编译时指定编译驱动器启动选项 -finfo，以便生成绝对模块文件 (.x30)。

1. 监视信息是指在指定编译选项 “-finfo”后，由 NC30 生成的信息。

1.3 注意

瑞萨科技的产品不是为在可能和人命相关的环境下使用的设备或者系统而设计和制造的产品。在研讨将本资料所记载的产品用于运输、交通车辆、医疗、航天、原子能控制、海底中继器的设备或者系统等特殊用途时，请与瑞萨科技、瑞萨解决方案或者经授权的瑞萨半导体产品经销商联系。

1.3.1 有关编译器升级版的注意事项

由 NC30 生成的机器语言指令 （汇编语言）的内容，视编译时指定的启动选项、升级版内容等而有所不同。因此，在更改启动选项或升级编译器版本后，确保重新评价应用程序的操作。

此外，当在中断处理与非中断处理例程之间，或在实时 OS 中的任务之间引用相同的 RAM 数据 （并更改其内容）时，总是确保行使专属的控制，如 volatile 的指定。同时，也为具有不同成员名称，但却映射到相同的 RAM 的位字段结构行使专属的控制。

1.3.2 有关 M16C 的类型独有部分的注意事项

当在 SFR 区中写入或读取寄存器时，有时可能需要使用一个特定的指令。由于这个特定的指令依不同的 MCU 类型而异，请参考您 MCU 的用户手册以了解详细信息。在这里，使用 ASM 函数将指令直接写入到程序中。

在本编译器中，可生成无法使用的指令，以写入及读取 SFR 区的寄存器。当以 C 语言读取 SFR 区中的寄存器时，无论编译器是何版本及是否使用了优化选项，确保和使用 asm 函数时一样，生成相同的正确指令。

当您如下列范例所示的 C 语言描述，对 SFR 区进行描述时，由于中断请求位不正常，本编译器所生成的汇编器代码将执行不想要的操作。

```
#pragma  ADDRESS  TA0IC  006Ch                /* M16C/60 MCU 的计时器 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 TA0IC.IR 变成 1 */
        {
                ;
        }
        TA0IC.IR = 0;                         /* 当它变成 1 时返回 0 到 TA0IC.IR */
}
```

图 1.2 对 SFR 区的 C 语言描述

1.4 范例程序开发

图 1.3 显示使用 NC30 开发范例程序的流程。下面是有关该程序的描述。
（项目 [1] 到 [4] 与图 1.3 中的相同数字对应）

(1) C 源程序 AA.c 使用 NC30 来进行编译，然后使用 as30 来进行汇编，以建立可再定位目标文件 AA.r30。
(2) 包含段信息的启动程序 ncrt0.a30 和包含文件 sect30.inc，通过改变段映射、段大小和中断向量表设置来匹配到系统。
(3) 对经修改的启动程序进行汇编，以建立可再定位的目标文件 ncrt0.a30。
(4) 通过从 nc30 运行的连接编辑器 ln30，连接两个可再定位的目标文件 AA.r30 和 ncrt0.a30，以建立绝对模块文件 AA.x30。

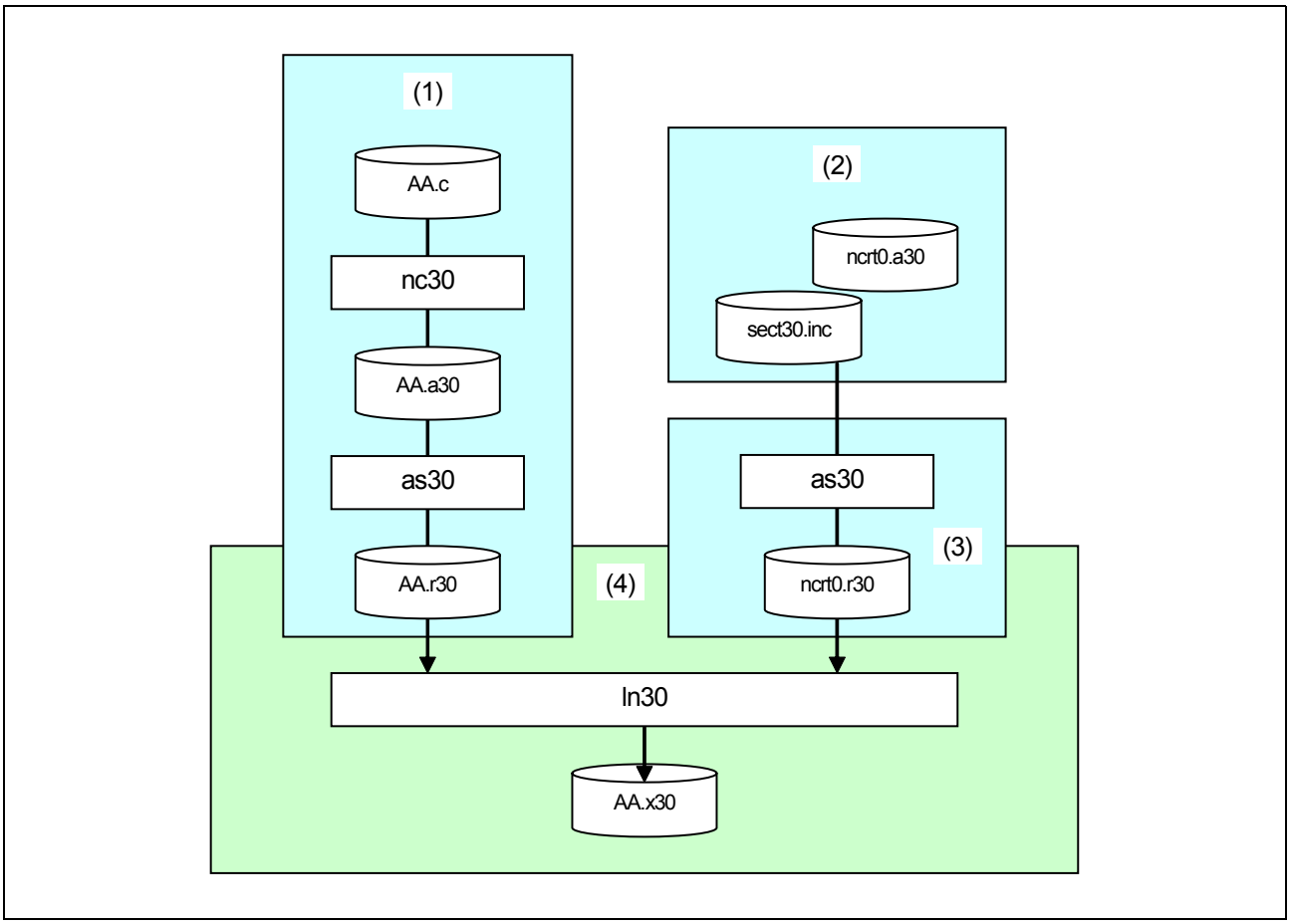


图 1.3　程序开发流程

图 1.3 是一个包含一系列在图 1.4 中显示的操作的命令描述文件范例。

```
AA.x30 : ncrt0.a30 AA.r30
        nc30 -oAA ncrt0.r30 AA.r30

ncrt0.r30 : ncrt0.a30
        as30 ncrt0.a30

AA.r30 : AA.c
        nc30 -c AA.c
```

图 1.4 命令描述文件范例

图 1.5 显示 NC30 执行图 1.4 所显示命令描述文件中的相同操作所需的命令行。

```
% nc30  -oAA  ncrt0.a30  AA.c<RET>

%: 表示提示符
<RET>: 表示回车键

* 连接时先指定 ncrt0.a30。
```

图 1.5 NC30 命令行的范例

1.5 NC30 输出文件

本章介绍在使用 NC30 和汇编语言源程序来编译样品程序 sample.c 时， C 源程序所输出的预处理结果。

1.5.1 输出文件简介

使用所指定的命令行选项， nc30 编译驱动器输出在图 1.6 中所显示的文件。在对图 1.7 所示的 C 源文件 smp.c 进行编译、汇编及连接时，文件输出的内容如下所示。

若要了解 as30 和 ln30 所输出的可再定位目标文件 （扩展名为 .r30）、打印文件 （扩展名为 .lst）及映射文件 （扩展名为 .map），请参考 AS30 的用户手册。

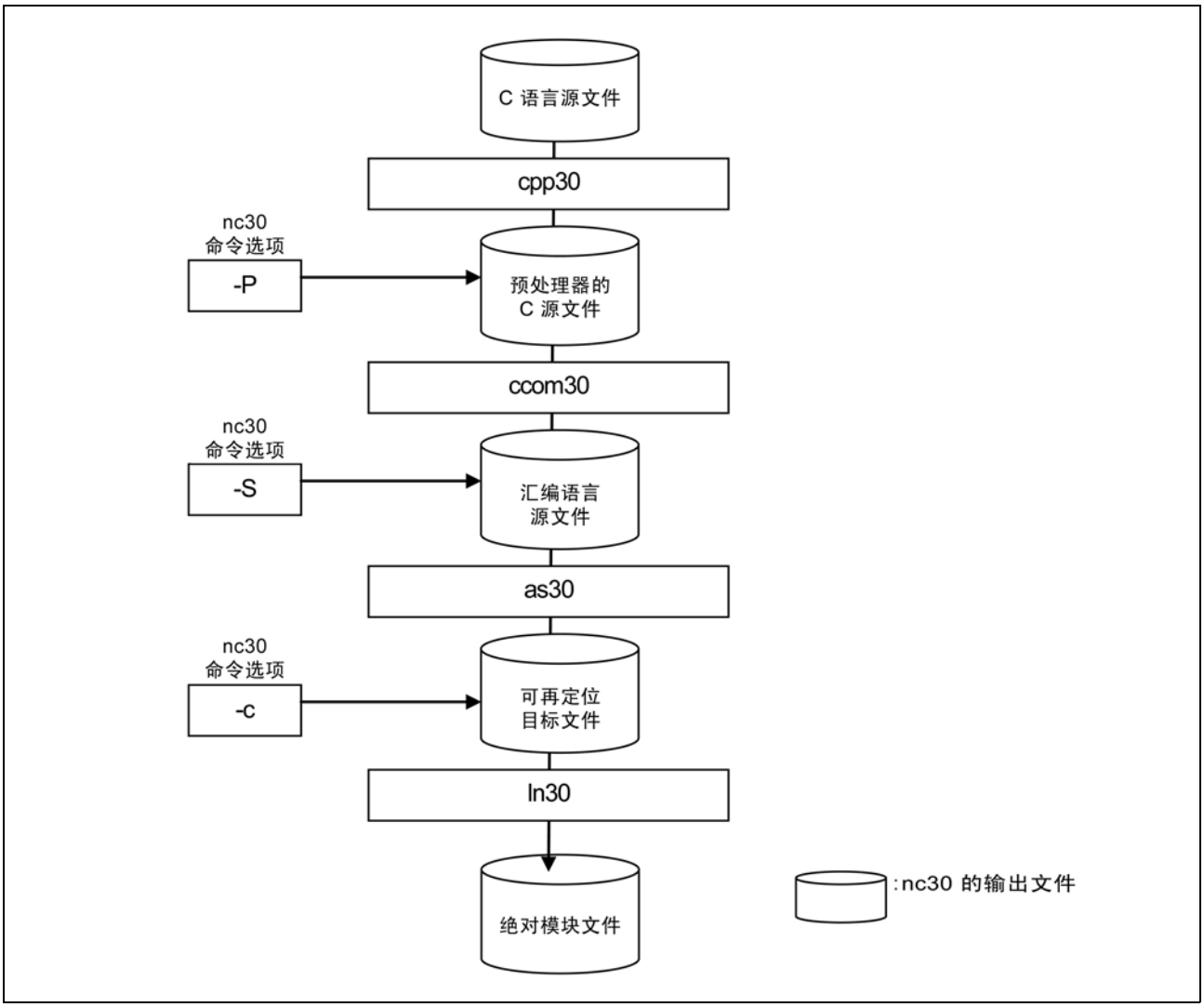


图 1.6 NC30 命令行选项与输出文件的关系

```
#include <stdio.h>
#define CLR  0
#define PRN  1

void        main(void)
{
            int         flag;

            flag = CLR;
#ifdef PRN
            printf( "flag = %d\n", flag );
#endif
}
```

图 1.7 范例 C 源文件 (sample.c)

1.5.2 经预处理的 C 源文件

cpp3 处理以 # 开头的预处理命令。这类操作包括标题文件内容、宏扩展及对条件编译的判断。

预处理器所输出的 C 源文件，包括了 C 源文件的 cpp30 处理结果。因此，请勿包含 #pragma 及 #line 以外的预处理行。您可以参考这些文件，以检查由编译器处理的程序的内容。文件的扩展名是 .i。

图 1.8 和图 1.9 是文件输出的范例。

```
typedef struct _iobuf {                                                    (1)
            char        _buff;
            int         _cnt;
            int         _flag;
            int         _mod;
            int         (*_func_in)(void);
            int         (*_func_out)(int);
} FILE;
                :
            （已省略）
                :
typedef     long                    fpos_t;

typedef     unsigned int            size_t;

extern FILE_iob[];
```

图 1.8 经预处理的 C 源文件的范例 (1)

```
extern int  getc(FILE _far *);                                                    (1)
extern int  getchar(void);
extern int  putc(int, FILE _far *);
extern int  putchar(int);
extern int  feof(FILE _far *);
extern int  ferror(FILE _far *);
extern int  fgetc(FILE _far *);
extern char _far *fgets(char _far *, int, FILE _far *);
extern int  fputc(int, FILE _far *);
extern int  fputs(const char _far *, FILE _far *);
extern size_tfread(void _far *, size_t, size_t, FILE _far *);
                :
            （已省略）
                :
extern int  printf(const char _far *, ...);
extern int  fprintf(FILE _far *, const char _far *, ...);
extern int  sprintf(char _far *, const char _far *, ...);
                :
            （已省略）
                :
extern int  init_dev(FILE _far *, int);
extern int  speed(int, int, int, int);
extern int  init_prn(void);
extern int  _sget(void);
extern int  _sput(int);
extern int  _pput(int);
extern const char _far *_print(int(*)(), const char _far *, int _far * _far *, int _far *);
```

```
void        main(void)                                                            (2)
{
            int         flag;

            flag = 0 ;                              ← (3)

            printf( "flag = %d\n", flag );          ← (4)

}
```

图 1.9　　经预处理的 C 源文件的范例 (2)

经预处理的源文件的内容如下：项目 (1) 到 (4) 与图 1.8 和图 1.9 中的 (1) 到 (4) 对应。

(1) 显示在 #include 中指定的标题文件 stdio.h 的扩展。

(2) 显示由扩展宏生成的 C 源程序。

(3) 显示在 #define 中指定的 CLR 被扩展为 0。

(4) 显示由于 PRN 在 #define 中指定为 1，所以符合编译条件，并输出 printf 函数。

1.5.3　　汇编语言源文件

汇编语言源文件是编译器 ccom30 转换 C 源文件的预处理结果所产生的文件，可由 AS30 进行处理。输出文件是扩展名为 .a30 的扩展语言源文件。

图 1.10 和图 1.11 是输出文件的范例。当指定 NC30 命令行选项 “-dsource （-dS）” 后， C 源文件的内容将作为注释包含在汇编语言源文件中。

```
                ._LANG    'C','X.XX.XX.XXX','REV.X'

    ;## NC30 C Compiler OUTPUT
    ;## ccom30 Version X.XX.XX.XXX
    ;## Copyright(C) XXXX(XXXX). Renesas Technology Corp.
    ;## and Renesas Solutions Corp., All Rights Reserved.
    ;## Compile Start Time XXX XXX XX XX:XX:XX XXXX

    ;## COMMAND_LINE: ccom30  C:\Renesas\nc30wa\v521r00TMP\sample.i -o .\sample.a30 -dS

    ;## Normal Optimize OFF                                                    (1)
    ;## ROM size OptimizeOFF
    ;## Speed Optimize  OFF
    ;## Default ROM is   far
    ;## Default RAM is   near

                .GLB        __SB__
                .SB         __SB__
                .FB         0

    ;## #       FUNCTION main
    ;## #       FRAME    AUTO      (    flag)    size  2,     offset -2
    ;## #       ARG Size(0)Auto Size(2)Context Size(5)


                .SECTIONprogram,CODE,ALIGN
                ._file      'sample.c'
                .align
                ._line      6
    ;## # C_SRC :{
                .glb        _main
    _main:
                enter       #02H
                ._line      9
    ;## # C_SRC :          flag = CLR;
                mov.w       #0000H,-2[FB];  flag
                ._line      11
    ;## # C_SRC :          printf( "flag = %d\n", flag );                  ← (2)
                push.w      -2[FB]      ;  flag
                push.l      #___T0
                jsr         _printf
                add.l       #06H,SP
                ._line      13
    ;## # C_SRC :}
                exitd
                    :
                （已省略）
                    :       .glb        _puts
                .glb        $ungetc
                .glb        _printf
                .glb        _fprintf
                .glb        _sprintf
                    :
                （已省略）
                    :
```

图 1.10　　汇编语言源文件的范例 (1) “sample.a30”

```
        .SECTIONrom_FO,ROMDATA
___T0:
        .byte   66H     ; 'f'
        .byte   6cH     ; 'l'
        .byte   61H     ; 'a'
        .byte   67H     ; 'g'
        .byte   20H     ; ' '
        .byte   3dH     ; '='
        .byte   20H     ; ' '
        .byte   25H     ; '%'
        .byte   64H     ; 'd'
        .byte   0aH
        .byte   00H
        .END

;## Compile End Time XX XXX XX XX:XX:XX XXXX
```

图 1.11 汇编语言源文件的范例 (2) “sample.a30”

汇编语言源文件的内容如下：项目 (1) 到 (2) 与图 1.10 中的 (1) 到 (2) 对应。

(1) 显示优化选项的状态，及 ROM 和 RAM 属性中 near 及 far 的初始设置的信息。

(2) 当指定 NC30 命令行选项 “-dsource (-dS)”后， C 源文件的内容将作为注释显示。

2. 使用编译器的基本方法

本章将描述如何启动编译驱动器 nc30 及有关的命令行选项。

2.1 启动编译器

2.1.1 nc30 命令格式

nc30 编译驱动器会启动编译器命令 （cpp30 和 ccom30）、汇编器命令 as30 及连接器命令 ln30，以建立一个绝对模块文件。若要启动 nc30，必须使用下列信息 （输入参数）：

1. C 源文件
2. 汇编语言源文件
3. 可再定位的目标文件
4. 命令行选项 （可选）

这些项目都将在命令行上指定。

图 2.1 所显示的是命令行格式。图 2.2 则显示一个范例。在该范例中执行的操作如下：

1. 对启动程序 ncrt0.a30 进行汇编。
2. 对 C 源程序 sample.c 进行编译和汇编。
3. 连接可再定位的目标文件 ncrt0.r30 和 sample.r30。

建立绝对模块文件 sample.x30。将使用下列命令行选项：

- 指定机器语言数据文件 sample.x30..... 选项 -o
- 指定汇编时列表文件 （扩展名为 .lst）的输出..... 选项 -as30 “-l”
- 指定连接时映射文件 （扩展名为 .map）的输出..... 选项 -ln30 “-ms”

```
% nc30Δ[ 命令行选项 ]Δ[ 汇编语言源文件名称 ]Δ[ 可再定位的目标文件名称 ]Δ<C 源文件名称 >

%　：　提示符
< >　：　强制性项目
[ ]　：　可选项目
Δ　：　空格
```

图 2.1　nc30 命令行格式

```
% nc30 -osample -as30 "-l" -ln30 "-ms" ncrt0.a30 sample.c<RET>

<RET>　：　回车键
* 连接时总是先指定启动程序。
```

图 2.2　nc30 命令行的范例

2.1.2 命令文件

编译驱动器可编译写有多个命令选项且加载到主机上的单一命令文件。

命令文件的使用可帮助克服 PC 等所施加的命令行字符数的限制。

(a) 命令文件输入格式

```
% nc30Δ[ 命令行选项]Δ<@ 文件名 >[ 命令行选项]

%    :  提示符
< >  :  强制性项目
[ ]  :  可选项目
Δ    :  空格
```

图 2.3　命令文件命令行格式

```
% nc30  -c  @test.cmd  -g<RET>

<RET>  :  回车键
* 连接时总是先指定启动程序。
```

图 2.4　ncrt0.a30<CR> 命令文件命令行的范例

命令文件以下面所述的形式编写。

```
命令文件描述                          ncrt0.a30<CR>
                                      sample1.c  sample2.r30<CR>
                  ──────────►         -g  -as30  -l<CR>
                                      -o<CR>
<CR>  :  表示回车。                   sample<CR>
```

图 2.5　命令文件描述的范例

(b) 命令文件描述的规则

下面是命令文件描述的应用规则：

- 一次只能指定一个命令文件。不可以同时指定多个命令文件。
- 不可在另一个命令文件中指定另一个命令文件。
- 命令文件中可编写多个命令行。
- 命令文件的新行字符以空格字符替代。
- 可在命令文件的一行中编写的最大字符数为 2,048。超出这个限制将发生错误。

(c) 使用命令文件时必须遵守的注意事项

可为命令文件名指定一个目录路径。若文件不存在于指定的目录路径中，则会发生错误。

将自动为 ln30 生成文件扩展名为 “.cm$”的命令文件，以便在连接时指定。因此，若存在文件扩展名为 “.cm$”的现有文件，现有文件将被盖写。请勿将文件扩展名为 “.cm$”的文件用于本编译器。不可以同时指定两个或多个命令文件。

若指定多个文件，编译器将显示 “Too many command files”（太多命令文件）的错误信息。

2.1.3 有关 NC30 命令行选项的注意事项

(a) 有关 nc30 命令行选项的编码的注意事项

nc30 命令行选项的编写有大小写区分。若以错误的大小写来编写，可能导致有些选项无效。

(b) 控制编译驱动器的选项的优先级

控制编译驱动器的选项的优先级。

```
-E      -P      -S      -c
← 高      优先级      低 →
```

因此，若同时指定了下列两个选项，例如，

- “-c”：在建立可再定位的模块文件 （扩展名为 .r30）后完成处理
- “-S”：在建立汇编语言源文件 （扩展名为 .a30）后完成处理， -S 选项具有优先级 。这即是说，编译驱动器将不会在汇编后进行任何进一步的处理。

因此，它将只生成一个汇编语言源文件。若要同时建议一个可再定位的文件和一个汇编语言源文件，可以使用 “-dsource （缩写为 -dS）”的选项。

2.1.4 nc30 命令行选项

(a) 用于控制编译驱动器的选项

表 2.1 显示用于控制编译驱动器的命令行选项。有关各个可选附注的详细信息，请参考附录 A。

表 2.1 用于控制编译驱动器的选项

选项	功能
-c	建立可再定位的模块文件 （扩展名为 .r30）并结束处理。[1]
- D *标识符*	定义标识符。与 #define 功能相同。
-dsource （缩写 -dS）	生成汇编语言源文件，并将其中的 C 语言源列表输出变为注释 （扩展名为 “.a30”）。（汇编后也不会将其删除。）
-dsource_in_list （缩写 -dSL）	除 “-dsource”的功能外，可生成汇编语言列表文件 (.lst)。
-E	仅调用预处理命令，并将结果输出为标准输出。
-I *目录*	指定包含在 #include 中指定的文件的目录。您最多可以指定 50 个目录。
-P	仅调用预处理命令并建立一个文件 （扩展名为 .i）。
-S	建立汇编语言源文件 （扩展名为 .a30）并结束处理。
-silent	禁止启动时的版权信息显示。
-U *预定义的宏*	取消指定的预定义宏。

(b) 指定输出文件的选项

表 2.2 显示指定输出机器语言数据文件的名称的命令行选项。

表 2.2 指定输出文件的选项

选项	功能
-dir *目录名称*	指定 ln30 生成的文件 （绝对模块文件、映射文件等）的目标目录。
-o *文件名*	指定 ln30 生成的文件 （绝对模块文件、映射文件等）的名称。此选项也可用于指定目标目录。 不要指定文件扩展名。

[1] 若未指定 -c、 -E、 -P 或 -S 的命令行选项， nc30 将在 ln30 完成处理后，输出已建立的文件，包括绝对加载模块文件 （扩展名为 .x30）在内。

(c) 版本与命令行信息显示选项

表 2.3 显示交叉工具版本数据和命令行信息的命令行选项。

表 2.3 显示版本数据和命令行信息的选项

选项	功能
-v	在执行期间显示命令程序的名称和命令行。
-V	显示编译器程序的启动信息，然后完成处理 （不编译）。

(d) 用于调试的选项

表 2.4 显示用于为 C 源文件输出符号文件的命令行选项。

表 2.4 用于调试的选项

选项	功能
-g	将调试信息输出到一个汇编器源文件 （扩展名为 .a30）。因此，能够执行 C 语言级的调试。
-genter	调用函数时，总是输出 enter 指令。 在使用调试器的堆栈跟踪功能时，确保指定此选项。
-gno_reg	禁止输出寄存器变量的调试信息。
-gold	此选项将以 Rev.E 格式输出调试信息。 指定此选项后，将自动指定 “-gno_reg”选项和 “-fauto_128”选项。

(e) 优化选项

表 2.5 显示用于优化程序执行速度和 ROM 容量的命令行选项。

表 2.5　　优化选项

选项	缩写	功能
-O[1-5]	无	优化速度和 ROM 大小。
-OR	无	优化 ROM 大小。
-OS	无	优化速度。
-OR_MAX	-ORM	执行 ROM 大小优先的优化。
-OS_MAX	-OSM	执行速度优先的优化。
-Ocompare_byte_to_word	-OCBTW	在相邻的地址上，以字的格式来比较字节数据。
-Oconst	-OC	通过常数替换 （在 const 修饰符的声明中）外部变量的引用来执行优化。
-Ofloat_to_inline	-OFTI	扩展直接插入的浮点运行时程序库，以加速浮点运算的处理。（仅适用于比较和乘法）
-Oforward_function_to_inline	-OFFTI	扩展所有的直接插入函数。
-Oglb_jmp	-OGJ	优化全局跳转。
-Oloop_unroll[= *循环计数*]	-OLU	按照循环计数的次数展开代码，而无须让循环语句循环出现。“循环计数”可以省略。省略 “循环计数”后，此选项将应用最多 5 次的循环计数。
-Ono_asmopt	-ONA	禁止启动汇编优化器 “aopt30”。
-Ono_bit	-ONB	根据位操作的组合来禁止优化。
-Ono_break_source_debug	-ONBSD	禁止会影响源数据的优化。
-Ono_float_const_fold	-ONFCF	禁止浮点数的常数合并处理。
-Ono_logical_or_combine	-ONLOC	禁止将连续的 OR 放在一起的优化。
-Ono_stdlib	-ONS	禁止标准程序库函数的直接插入填充和程序库函数的修改。
-Osp_adjust	-OSA	优化堆栈校正码的删除。这可减少 ROM 所需的容量。然而，这可能导致所使用的堆栈数增加。
-Ostack_frame_align	-OSFA	在每个边界对齐堆栈帧。
-Ostatic_to_inline	-OSTI	静态函数被当作直接插入函数处理。
-O5OA	无	在选择优化选项 “-O5”后，禁止根据位操作指令的代码生成。

(f) 修改生成的代码的选项

表 2.6 到表 2.7 显示用于控制 nc30 所生成的汇编代码的命令行选项。

表 2.6 修改生成的代码的选项 (1)

选项	缩写	功能
-fansi	无	使 “-fnot_reserve_far_and_near”、“-fnot_reserve_asm”和 “-fextend_to_int”有效。
-fchar_enumerator	-fCE	将 enumerator 类型作为 unsigned char 类型处理，而不是作为 int 类型处理。
-fconst_not_ROM	-fCNR	不会将 const 指定的类型作为 ROM 数据处理。
-fdouble_32	-fD32	本选项指定和 float 类型一样，以 32 位的数据长度处理 double 类型。
-fenable_register	-fER	使寄存器存储类可用。
-fextend_to_int	-fETI	在将 char 类型的数据扩展为 int 类型后执行操作。（根据 ANSI 的标准来扩展。）[2]
-ffar_RAM	-fFRAM	将 RAM 数据的默认属性更改为 far。
-finfo	无	将 “STK Viewer”、“Map Viewer”和 “utl30”所需的信息输出到绝对模块文件 (.x30)。
-fJSRW	无	将调用函数的默认指令更改为 JSR.W。
-fbit	-fB	生成代码并基于假设可以通过使用绝对寻址，对映射到 near 区域的所有外部变量按位操作指令执行。
-fno_carry	-fNC	当使用 far 类型指针间接存取数据时，禁止进位标志的加法。
-fauto_128	-fA1	将可用的堆栈帧限制为 128 字节。
-ffar_pointer	-fFP	将 pointer 类型变量的默认属性更改为 far。
-fnear_ROM	-fNROM	将 ROM 数据的默认属性更改为 near。
-fno_align	-fNA	不对齐函数的起始地址。
-fno_even	-fNE	将所有数据分配到奇数段，输出时不分开奇数数据与偶数数据。
-fno_switch_table	-fNST	指定此选项时，在进行比较后发生转移的代码将生成到一个 switch 语句中。
-fnot_address_volatile	-fNAV	不将 #pragma ADDRESS (#pragma EQU) 指定的变量视为 volatile 指定的变量。
-fnot_reserve_asm	-fNRA	保留字不包括 asm。（只有 _asm 有效。）
-fnot_reserve_far_and_near	-fNRFAN	保留字不包括 far 和 near。（只有 _far 和 _near 有效。）
-fnot_reserve_inline	-fNRI	保留字不包括 far 和 near。（只有 _inline 作为保留字。）
-fsmall_array	-fSA	参考 far 类型的数组时，如果在编译时不知数组总大小，此选项将假设数组总大小低于 64 Kb，并以 16 位计算下标。
-fswitch_other_section	-fSOS	此选项将 ‘switch’语句的 ROM 表输出至其它段，而不是程序段。
-fchange_bank_always	-fCBA	此选项允许您将多个变量写入扩展区。
-fauto_over_255	-fAO2	将每个函数可保留的堆栈帧大小更改为 64K 字节。

2. 在 ANSI 的规则下求值所得的 char 类型数据或 signed char 类型数据，将总是被扩展为 int 类型数据。
这是因为不这么做的话，对 char 类型的运算 （例如 c1=c2*2/c3）将发生溢出，从而无法获取所要的结果。

表 2.7　修改生成的代码的选项 (2)

选项	缩写	功能
-fsizet_16	-fS16	将 size_t 的类型定义从 unsigned long 类型更改为 unsigned int 类型。
-fptrdifft_16	-fP16	将 ptrdiff_t 的类型定义从 signed long 类型更改为 signed int 类型。
-fuse_DIV	-fUD	此选项将更改除法运算的生成代码。
-fuse_MUL	-fUM	此选项将更改乘法运算的生成代码。
-R8C	无	生成 R8C/Tiny 系列的目标代码。
-R8CE	无	生成适用于 R8C/Tiny （ROM 64K 版本）系列的代码。

(g)　程序库的指定选项

表 2.8 列出可用来指定一个程序库文件的启动选项。

表 2.8　程序库的指定选项

选项	功能
-l *程序库文件名*	指定 ln30 在连接文件时使用的程序库文件。

(h) 警告选项

表 2.9 显示在违反 nc30 语言的指定时输出警告信息的命令行选项。

表 2.9　警告选项

选项	缩写	功能
-Wall	无	为所有可检测的警告显示信息。 （不包括 -Wlarge_to_small 和 “-Wno_used_argument”所输出的警报）
-Wccom_max_warnings = *警告计数*	-WCMW	此选项允许您指定由 ccom30 输出的警告信息数的上限。
-Werror_file< *文件名* >	-WEF	将错误信息输出至指定的文件。
-Wlarge_to_small	-WLTS	按大小次序输出有关变量隐性转换的警告信息。
-Wmake_tagfile	-WMT	按照每一个源文件的标志文件输出错误和警告信息。
-Wnesting_comment	-WNC	对含有 “*/”的注释输出警告。
-Wno_stop	-WNS	防止出现错误时停止编译器。
-Wno_used_argument	-WNUA	为未使用的函数参数输出警告信息。
-Wno_used_function	-WNUF	在连接时显示未使用的全局函数。
-Wno_used_static_function	-WNUSF	显示不需要生成代码的静态函数名称。
-Wno_warning_stdlib	-WNWS	当已指定 “-Wnon_prototype”或 “-Wall”时，如果再指定此选项将会禁止 “Alarm for standard libraries which do not have prototype declaration”（没有原型声明的标准程序库的警告）。
-Wnon_prototype	-WNP	为不具有原型声明的函数输出警告信息。
-Wstdout	无	将错误信息输出至主机标准输出 (stdout)。
-Wstop_at_link	-WSAL	在连接时若发生警告，则停止连接源文件，以禁止绝对模块文件的生成。
-Wstop_at_warning	-WSAW	在编译时若发生警告，则停止编译源文件。
-Wundefined_macro	-WUM	对在 #if 中使用未定义的宏发出警告。
-Wuninitialize_variable	-WUV	对尚未初始化的自动变量输出警告。
-Wunknown_pragma	-WUP	对不支持的 #pragma 输出警告信息。

(i) 汇编和连接选项

表 2.10 显示指定 as30 和 ln30 的选项的命令行选项。

表 2.10　汇编和连接选项

选项	功能
-as30Δ< 选项 >	为 as30 连接命令指定选项。如果指定两个或多个选项，则用双引号将它们括起来。
-ln30Δ< 选项 >	为 ln30 汇编命令指定选项。如果指定两个或多个选项，则用双引号将它们括起来。

2.2 准备启动程序

为将 C 语言程序 “写”进 ROM 内， NC30 提供有一个用汇编语言编写的样品启动程序，以对硬件（M16C/60）进行初始设置、查找段及设置中断向量地址表等。这个启动程序需要进行修改，以适合所安装的器件使用。

下面提供有关启动程序的说明，并描述其制定方法。

2.2.1 启动样品程序

NC30 启动程序由下列两个文件所组成：

- ncrt0.a30
  编写一个在复位后立即执行的程序。
- sect30.inc
  ncrt0.a30 随附，这个文件用于定义段的位置 （存储器映射）。

图 2.6 到图 2.10 显示 ncrt0.a30 源程序列表。

```
;;-------------------------------------------------------------------
; HEEP SIZE 定义                                                    ← (1)
;-------------------------------------------------------------------
.if __HEAP__ == 1                    ; 用于 HEW

HEAPSIZE.equ        0h

.else
.if __HEAPSIZE__ == 0

HEAPSIZE.equ        300h

.else                                ; 用于 HEW

HEAPSIZE.equ        __HEAPSIZE__

.endif
.endif
```

(1) 定义堆大小。

图 2.6　启动程序列表 (1) (ncrt0.a30)

```
;------------------------------------------------------------------------
; STACK SIZE 定义                                                ← (2)
;------------------------------------------------------------------------
.if __USTACKSIZE__ == 0

STACKSIZE           .equ        300h

.else                                       ; 用于 HEW

STACKSIZE           .equ        __USTACKSIZE__

.endif

;------------------------------------------------------------------------
; INTERRUPT STACK SIZE 定义                                      ← (3)
;------------------------------------------------------------------------
.if __ISTACKSIZE__ == 0

ISTACKSIZE    .equ    300h

.else                           ; 用于 HEW

ISTACKSIZE    .equ    __ISTACKSIZE__

.endif

;------------------------------------------------------------------------
; INTERRUPT VECTOR ADDRESS  定义                                 ← (4)
;------------------------------------------------------------------------
VECTOR_ADR          .equ        0ffd00h
SVECTOR_ADR         .equ        0ffe00h

;------------------------------------------------------------
; special 页定义
;------------------------------------------------------------
;           special 页的宏定义
;
; 格式：
;           SPECIAL     编号
;

SPECIAL.macro       NUM
            .org        0FFFFEH-(NUM*2)
            .glb        __SPECIAL_@NUM
            .word       __SPECIAL_@NUM  & 0FFFFH
.endm
;------------------------------------------------------------------------
; 段分配
;------------------------------------------------------------------------
            .list OFF
            .include sect30.inc                                ← (5)
            .list ON
```

(2) 定义用户堆栈大小。
(3) 定义中断堆栈大小。
(4) 定义中断向量表的起始地址。
(5) 包含 sect30.inc

图 2.7　　启动程序列表 (2) (ncrt0.a30)

```
;=====================================================================
; 中断段起始
;---------------------------------------------------------------------
        .insf       start,S,0
        .glb        start
        .section    interrupt
start:                                                          ← (6)
;---------------------------------------------------------------------
; 复位后，程序将启动
;---------------------------------------------------------------------
        ldc         #istack_top,isp     ；设置 istack 指针
        mov.b       #02h,0ah
        mov.b       #00h,04h            ；设置处理器模式            ← (7)
        mov.b       #00h,0ah
        ldc         #0080h,    flg                                  ← (8)
        ldc         #stack_top,sp       ；设置堆栈指针
        ldc         #data_SE_top,       sb          ；设置 sb 寄存器

        fset        b                               ；转换到存储体 1
        ldc         #data_SE_top,       sb          ；设置 sb 寄存器
        fclr        b                               ；转换到存储体 0

        ldc         #VECTOR_ADR,intb

;=====================================================================
；NEAR 区初始化。
;---------------------------------------------------------------------
；bss 零清除                                                     ← (9)
;---------------------------------------------------------------------
        N_BZERObss_SE_top,bss_SE
        N_BZERObss_SO_top,bss_SO
        N_BZERObss_NE_top,bss_NE
        N_BZERObss_NO_top,bss_NO

;---------------------------------------------------------------------
；初始化数据段                                                   ← (10)
;---------------------------------------------------------------------
        N_BCOPYdata_SEI_top,data_SE_top,data_SE
        N_BCOPYdata_SOI_top,data_SO_top,data_SO
        N_BCOPYdata_NEI_top,data_NE_top,data_NE
        N_BCOPYdata_NOI_top,data_NO_top,data_NO
```

(6) 复位后，从这个标签 （start （开始））开始执行
(7) 设置处理器操作模式
(8) 设置 IPL 和每个标志。
(9) 清除 near bss 段 （至零）。
(10) 将 near 数据段和 SBDATA 数据段的初始值移到 RAM。

图 2.8　　启动程序列表 (3) (ncrt0.a30)

```
;=====================================================================
;  FAR 区初始化。
;---------------------------------------------------------------------
;  bss 零清除                                                  ← (11)
;---------------------------------------------------------------------

        BZERO       bss_FE_top,bss_FE
        BZERO       bss_FO_top,bss_FO

;---------------------------------------------------------------------
;  从 edata_EI(OI) 段复制 edata_E(O) 段                        ← (12)
;---------------------------------------------------------------------
        BCOPY       data_FEI_top,data_FE_top,data_FE
        BCOPY       data_FOI_top,data_FO_top,data_FO

        ldc         #stack_top,sp

;       .stk        -40

;=====================================================================
;  堆区初始化                                                  ← (13)
;---------------------------------------------------------------------
.if __HEAP__ != 1
        .glb        __mnext
        .glb        __msize
        mov.w       #(heap_top&0FFFFH), __mnext
        mov.w       #(heap_top>>16), __mnext+2
        mov.w       #(HEAPSIZE&0FFFFH), __msize
        mov.w       #(HEAPSIZE>>16), __msize+2
.endif
;=====================================================================
;  初始化标准 I/O                                              ← (14)
;---------------------------------------------------------------------
.if __STANDARD_IO__ == 1
        .glb        __init
        .call       __init,G
        jsr.a       __init
.endif

;=====================================================================
;  调用 main() 函数                                            ← (15)
;---------------------------------------------------------------------
        ldc         #0h,fb          ;  用于调试器

        .glb        _main
        jsr.a       _main
```

(11) 清除 far bss 段 （至零）。
(12) 将 far 数据段的初始值移到 RAM。
(13) 初始化堆区。若未使用任何存储器管理函数，则将这一行变为注释。
(14) 调用初始化标准 I/O 的 init 函数。若未使用任何 I/O 函数，则将这一行变为注释。
(15) 调用 ‘main’ 函数。
* 调用 ‘main’ 函数时，未允许中断。因此，使用中断函数时，
  通过 FSET 命令来允许中断。

图 2.9　　启动程序列表 (4) (ncrt0.a30)

```
;=====================================================================
; exit() 函数                                                  ← (16)
;---------------------------------------------------------------------
        .glb    _exit
        .glb    $exit
_exit:                                  ; 结束程序
$exit:
        jmp     _exit
        .einsf

;=====================================================================
; 虚设的中断函数                                               ← (17)
;---------------------------------------------------------------------
        .glb    dummy_int
dummy_int:
        reit
        .end
;
(16) 退出函数。
(17) 虚设的中断处理函数。
```

图 2.10 启动程序列表 (5) (ncrt0.a30)

2.2.2 定制启动程序

(a) 启动程序处理概述

(1) 关于 ncrt0.a30

这个程序在程序启动时或复位后立即运行。它主要执行下列处理：

- 设置 SBDATA 区 （它是使用 SB 相对寻址模式的存取区）的前列地址 (__SB__)。
- 设置处理器的操作模式。
- 初始化堆栈指针 （ISP 寄存器和 USP 寄存器）。
- 初始化 SB 寄存器。
- 初始化 INTB 寄存器。
- 初始化 near 数据区。

  bss_NE bss_NO bss_SE 和 bss_SO 段将被清除 （至 0）。

  同时， ROM 区 （data_NEI、 data_NOI、 data_SEI、 data_SOI）中的初始值将被转移到 RAM （data_NE、 data_NO、 data_SE 和 data_SO）。
- 初始化 far 数据区。

  bss_FE 和 bss_FO 段将被清除 （至 0）。

  同时， ROM 区 （data_FEI、 data_FOI）中存储的初始值将被转移到 RAM （data_FE、 data_FO）。
- 初始化堆区。
- 初始化标准 I/O 函数程序库。
- 初始化 FB 寄存器。
- 调用 ‘main’ 函数。

(b) 修改启动程序

图 2.11 摘录了修改启动程序以匹配目标系统所需采取的步骤。

图 2.11　修改启动程序的范例步骤

(c) 需要注意的启动程序修改范例

(1) 不使用标准 I/O 函数时的设置

init 函数[3] 将初始化 M16C/80 系列 I/O。它在 ncrt0.a30 中先于 main 被调用。
图 2.12 显示调用 init 函数的部分。

若应用程序不使用标准 I/O，则在 ncrt0.a30 中将 init 函数变为注释。

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

图 2.12　调用 init 函数的 ncrt0.a30 部分

若只是使用 sprintf 和 sscanf，则不需要调用 init 函数。

[3] init 函数也为标准 I/O 函数初始化单片机 （硬件）。默认情况下， M16C/60 和 R8C/Tiny 被假设是要进行初始化的单片机。
当使用标准 I/O 函数时，视使用单片机的系统而定，可能需要对 init 函数等进行修改。

(2) 不使用存储器管理函数时的设置

若要使用 calloc 和 malloc 等存储器管理函数，除了在堆段 （heap section）中分配一个区，也将在 ncrt0.a30 中进行下列设置。

1. 外部变量 char *_mnext 的初始化
   初始化 heap_top 标签，堆段的起始地址。
2. 外部变量 unsigned_msize 的初始化
   初始化在 “2.2.2 （e）堆段大小”中设置的 “HEAPSIZE”表达式。

图 2.13 显示在 ncrt0.a30 中执行的初始化。

```
;======================================================================
；堆区初始化
;----------------------------------------------------------------------
.if __HEAP__ != 1
        .glb        __mnext
        .glb        __msize
        mov.w       #(heap_top&0FFFFH), __mnext
        mov.w       #(heap_top>>16) __mnext+2
        mov.w       #(HEAPSIZE&0FFFFH), __msize
        mov.w       #(HEAPSIZE>>16), __msize+2
.endif
```

图 2.13　使用存储器管理函数时的初始化 （ncrt0.a30）

若不使用存储器管理函数，则将整个初始化段变为注释。这样做可以停止连接不需要的程序库项目，从而节省 ROM 的容量。

(3) 有关编写初始化程序的注意事项

当自行编写要添加到启动程序的初始化程序时，请注意下列事项。

1. 若初始化程序更改了 U 或 B 标志，请在退出初始化程序时将这些标志返回到原始状态。请勿更改 SB 寄存器的内容。
2. 若初始化程序调用一个以 C 编写的子例程，则必须注意下列两点：
   - 仅在清除 B 和 D 标志后调用 C 子例程。
   - 仅在设置 U 标志后调用 C 子例程。

(d) 设置堆栈段大小

一个堆栈段中包含用于用户堆栈的域及用于中断堆栈的域。由于一定会使用堆栈，所以请必定保留一个域。堆栈大小应设置为所要使用大小的最大值。[4]

使用堆栈大小计算工具 STK Viewer 来计算堆栈大小。

4. 堆栈也会在启动程序中使用。虽然初始值会在调用 main() 函数前被重新加载，但必须考虑到 main() 函数等所使用的堆栈大小可能不够的情况。

(e) 堆段大小

在程序中使用存储器管理函数 calloc 和 malloc 将堆设置为最大的存储器分配大小。若未使用这些存储器管理函数，则将堆设置为 0。确保堆段未超出物理 RAM 区。

```
;------------------------------------------------------------------------
;  HEEP SIZE 定义
;------------------------------------------------------------------------
.if __HEAP__ == 1                                   ;  用于 HEW

HEAPSIZE.equ        0h

.else
.if __HEAPSIZE__ == 0

HEAPSIZE.equ        300h

.else                                               ;  用于 HEW

HEAPSIZE.equ        __HEAPSIZE__

.endif
.endif
```

图 2.14　设置堆段大小的范例 （ncrt0.a30）

(f) 设置中断向量表

将中断向量表的前列地址设置到图 2.15 中的 ncrt0.a30 部分。INTB 寄存器由中断向量表的前列地址进行初始化。

```
;------------------------------------------------------------------------
; INTERRUPT VECTOR ADDRESS  定义
;------------------------------------------------------------------------
VECTOR_ADR          .equ            0ffd00h
SVECTOR_ADR         .equ            0ffe00h
```

图 2.15　设置中断向量表的前列地址的范例 （ncrt0.a30）

样品启动程序具有为下表所设置的值。

0FFD00H - 0FFDFFH:	中断向量表
0FFE00H - 0FFFFFH:	special 页向量表和固定向量表

通常，这些设置值不需要进行修改。

(g) 设置处理器模式寄存器

在图 2.16 所显示的 ncrt0.a30 部分中的 04H 地址（处理器模式寄存器）上设置处理器操作模式，以匹配目标系统。

```
;-----------------------------------------------------------------------
；复位后，程序将启动
;-----------------------------------------------------------------------
        :
（已省略）
        :
  mov.b         #00h,04h            ；设置处理器模式
        :
（已省略）
        :
```

图 2.16　设置处理器模式寄存器的范例（ncrt0.a30）

参考所使用的单片机的用户手册，以获取有关处理器模式寄存器的详细信息。

2.2.3 定制 NC30 存储器映射

(a) 段的结构

在本地环境编译器中，由编译器生成的可执行文件将由操作系统，如 UNIX，映射到存储器。然而，若使用交叉环境的编译器，如本编译器，用户必须确定存储器映射。

使用本编译器，存储类变量、具有初始值的变量、没有初始值的变量、字符串数据、中断处理器和中断向量地址表等，将根据它们的功能，作为独立的段映射到单片机系列的存储器中。

段的名称由基本名称和属性所组成，如下所示：

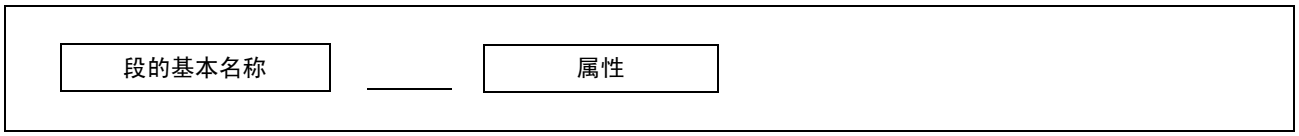


图 2.17　段的名称

表 2.11 显示段的基本名称，表 2.12 则显示属性。

表 2.11　段的基本名称

段的基本名称	内容
data	存储具有初始值的数据
bss	存储没有初始值的数据
rom	存储字符串及以 #pragma ROM 指定或具有 const 修饰符的数据。

表 2.12　段的命名规则

属性	含义		目标段的基本名称
I	包含数据初始值的段		data
N/F/S	N	near 属性 [5]	data、 bss、 rom
	F	far 属性	
	S	SBDATA 属性	data、 bss
E/O	E	偶数数据大小	data、 bss、 rom
	O	奇数数据大小	

表 2.13 显示上面所述命名规则以外的段的内容。

表 2.13　段的名称

段的名称	内容
fvector	这个段存储单片机的固定向量的内容。
heap	这个存储区在程序执行时由存储器管理函数 （如 malloc）动态分配。 这个段可以在单片机的 RAM 区中的任何位置分配。
program	存储程序
program_S	存储为其指定了 #pragma SPECIAL 的程序。
stack	这个区被用作一个堆栈。在 0400H 到 0FFFFH 之间的地址上分配这个区。
switch_table	switch 语句的转移表所被分配的段。这个段仅能使用 “-fSOS”选项来生成。
vector	这个段存储单片机的中断向量表的内容。可使用 intb 寄存器相对寻址在单片机的整个存储器空间的任何位置上分配中断向量表。 有关详细信息，请参考单片机用户手册。

这些段根据在启动程序包含文件 sect30.inc 中的设置映射到存储器。可通过修改包含文件来更改映射。

图 2.18 显示如何根据样品启动程序的包含文件 sect30.inc 将段映射。

5. near 和 far 是 NC30 的修饰符，用来说明寻址模式。
near..... 可在 000000H 到 00FFFFH 之间存取
far..... ... 可在 000000H 到 0FFFFFH 之间存取

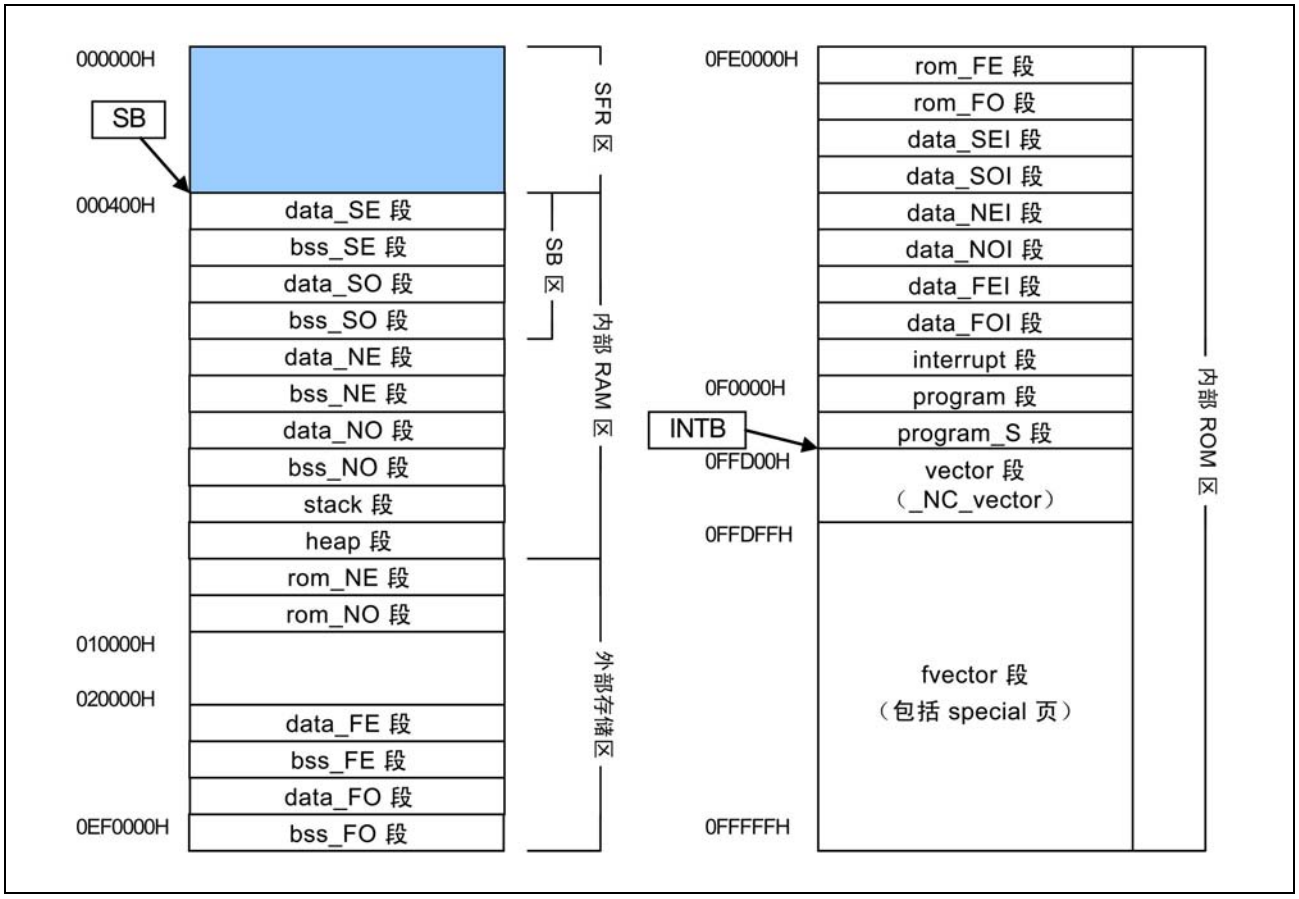


图 2.18　　段映射的范例

(b)　存储器映射设置文件的概要

(1)　关于 sect30.inc

这个程序包含于 ncrt0.a30。它主要执行下列处理：

- 映射各个段 （按顺序）
- 设置段的起始地址
- 定义堆栈和堆段的大小
- 设置中断向量表
- 设置固定向量表

(c) 修改 sect30.inc

图 2.19 总结了修改启动程序来匹配目标系统所需采取的步骤。

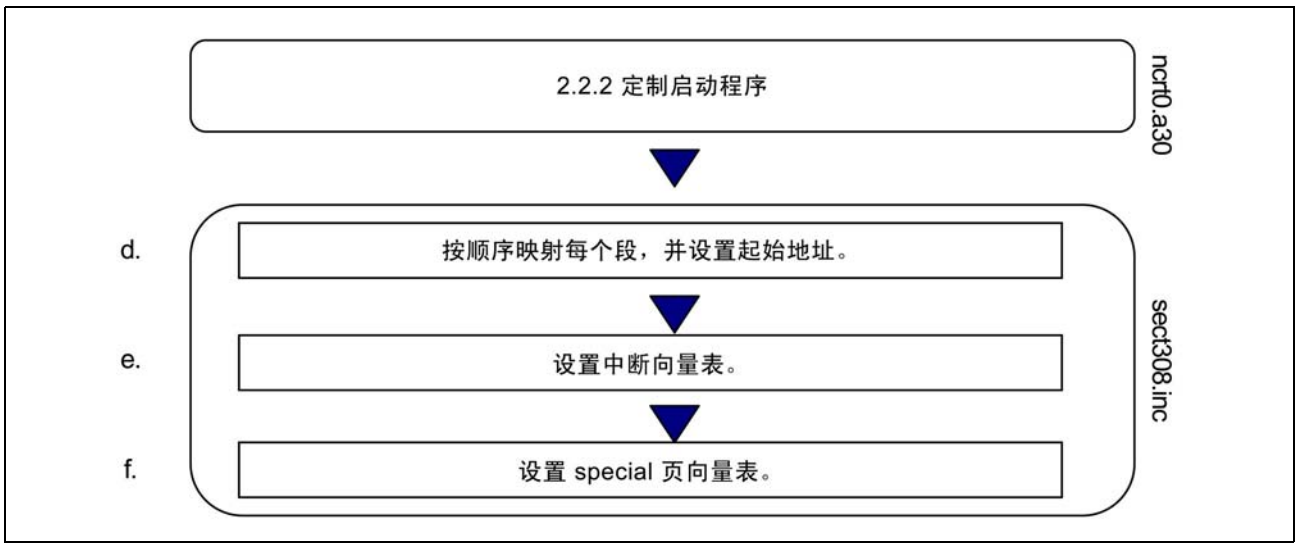


图 2.19　修改启动程序的范例步骤

(d) 映射和编排段的顺序及指定起始地址

在启动程序的 sect30.inc 包含文件中将段映射到存储器并编排顺序，然后指定它们的起始地址 （映射程序和数据到 ROM 和 RAM）。

段会根据在 sect30.inc 中定义的顺序映射到存储器。使用汇编器伪指令 .ORG 来指定它们的起始地址。

图 2.20 是这些设置的一个范例。

```
.section    program
.org                    0F0000H          ← 指定程序段的起始地址
```

图 2.20　设置段的起始地址的范例

若未对一个段指定任何起始地址，该段将跟随之前所定义的段进行映射。

(1) 将段映射到存储器的规则

由于单片机存储器属性 （RAM 和 ROM）的缘故，有些段只能映射到特定的区。将段映射到存储器时，请应用下列规则。

1. 映射到 RAM 的段

- stack 段
- data_SE 段
- data_NE 段
- bss_SE 段
- bss_NE 段
- bss_FE 段
- heap 段
- data_SO 段
- data_NO 段
- bss_SO 段
- bss_NO 段
- bss_FO 段

2. 映射到 ROM 的段

- program 段
- fvector 段
- rom_NO 段
- rom_FO 段
- data_SOI 段
- data_NOI 段
- data_FOI 段
- interrupt 段
- rom_NE 段
- rom_FE 段
- data_SEI 段
- data_NEI 段
- data_FEI 段

也请注意某些段只能映射到单片机存储器空间的特定存储区。

1. 只能映射到 0H - 0FFFFH （near 区）的段

- data_NE 段
- data_SE 段
- bss_NE 段
- bss_SE 段
- rom_NE 段
- stack 段
- data_NO 段
- data_SO 段
- bss_NO 段
- bss_SO 段
- rom_NO 段

2. 只能映射到 0F0000H - 0FFFFFH 的段

- program_S 段
- fvector 段

3. 在 M16C/60 系列中可映射到任何区的段。

- program 段
- data_NEI 段
- data_FE 段
- data_FEI 段
- data_SEI 段
- bss_FE 段
- rom_FE 段
- vector 段
- data_NOI 段
- data_FO 段
- data_FOI 段
- data_SOI 段
- bss_FO 段
- rom_FO 段

若下列任何数据段的大小为 0，则不须定义它们。

- data_SE 段
- data_SO 段
- data_NE 段
- data_NO 段
- data_FE 段
- data_FO 段
- bss_NE 段
- bss_FE 段
- bss_SE 段
- rom_NE 段
- rom _FE 段
- data_SEI 段
- data_SOI 段
- data_NEI 段
- data_NOI 段
- data_FEI 段
- data_FOI 段
- bss_NO 段
- bss_FO 段
- bss_SO 段
- rom _NO 段
- rom_FO 段

(2) 单芯片模式中的段映射范例

图 2.21 到图 2.24 是用于在单芯片模式中将段映射到存储器的 sect30.inc 包含文件的范例。

```
;------------------------------------------------------------
;
;           段的排列
;
;------------------------------------------------------------
；Near RAM 数据区
;------------------------------------------------------------
；SBDATA 区
            .section        data_SE,DATA
            .org            400H
data_SE_top:

            .section        bss_SE,DATA,ALIGN
bss_SE_top:

            .section        data_SO,DATA
data_SO_top:

            .section        bss_SO,DATA
bss_SO_top:

；Near RAM 区
            .section        data_NE,DATA,ALIGN
data_NE_top:

            .section        bss_NE,DATA,ALIGN
bss_NE_top:

            .section        data_NO,DATA
data_NO_top:

            .section        bss_NO,DATA
bss_NO_top:

;------------------------------------------------------------
；堆栈区
;------------------------------------------------------------
            .section        stack,DATA,ALIGN
            .blkb           STACKSIZE
            .align
stack_top:

            .blkb           ISTACKSIZE
            .align
istack_top:
```

图 2.21　　sect30.inc 在单芯片模式中的列表 (1)

```
;------------------------------------------------------------------
;           heap 段
;------------------------------------------------------------------
.if __HEAP__ != 1
            .section        heap,DATA
heap_top:
            .blkb           HEAPSIZE
.endif

;------------------------------------------------------------------
;  Near ROM 数据区
;------------------------------------------------------------------
            .section        rom_NE,ROMDATA,ALIGN
rom_NE_top:

            .section        rom_NO,ROMDATA
rom_NO_top:

;------------------------------------------------------------------
;  Far RAM 数据区                                          ← 因不必要，可删除此部分。
;------------------------------------------------------------------
;                                                           在这种情况下，需要删除 ncrt0.a30
            .section        data_FE,DATA                    far 区域中的初始化程序。
            .org                        XXXX0H
data_FE_top:

            .section        bss_FE,DATA,ALIGN
bss_FE_top:

            .section        data_FO,DATA
data_FO_top:

            .section        bss_FO,DATA
bss_FO_top:
```

图 2.22　　sect30.inc 在单芯片模式中的列表 (2)

```
;------------------------------------------------------------
;  Far ROM 数据区
;------------------------------------------------------------
            .section        rom_FE,ROMDATA
            .org                        F0000H
rom_FE_top:

            .section        rom_FO,ROMDATA
rom_FO_top:

;------------------------------------------------------------
;  ‘data’段的初始数据
;------------------------------------------------------------:

            .section        data_NEI,ROMDATA
data_NEI_top:

            .section        data_NOI,ROMDATA
data_NOI_top:

            .section        data_FEI,ROMDATA
data_FEI_top:

            .section        data_FOI,ROMDATA
data_FOI_top:

;------------------------------------------------------------
;  代码区
;------------------------------------------------------------
            .section        interrupt,ALIGN

            .section        program,ALIGN

            .section        program_S

.if         __MVT__ == 0
;------------------------------------------------------------
;  变量向量段
;------------------------------------------------------------
            .section        vector,ROMDATA              ;  变量向量表
            .org            VECTOR_ADR
                :
            （已省略）
                :
            .lword          dummy_int                   ;  软件 int 63
.else       ; __MVT__
            .section        __NC_rvector,ROMDATA
            .org            VECTOR_ADR
.endif      ; __MVT__
```

图 2.23　　sect30.inc 在单芯片模式中的列表 (3)

```
.if     __MST__ == 0
;=====================================================================
; 固定向量段
;---------------------------------------------------------------------
        .section    svector,ROMDATA                 ; special 页向量表
        .org            SVECTOR_ADR
;=====================================================================
; special 页定义
;---------------------------------------------------------------------
;       宏在 ncrt0.a30 中定义
;       格式：SPECIAL 编号
;
;---------------------------------------------------------------------
;       SPECIAL 255
            :
        （已省略）
            :
;       SPECIAL 18
;
.else   ; __MST__
        .section    __NC_svector,ROMDATA
        .org            SVECTOR_ADR
.endif  ; __MST__
;=====================================================================
; 固定向量段
;---------------------------------------------------------------------
        .section    fvector,ROMDATA
        .org        0FFFDCH
UDI:
        .lword      dummy_int
OVER_FLOW:
        .lword      dummy_int
BRKI:
        .lword      dummy_int
ADDRESS_MATCH:
        .lword      dummy_int
SINGLE_STEP:
        .lword      dummy_int
WDT:
        .lword      dummy_int
DBC:
        .lword      dummy_int
NMI:
        .lword      dummy_int
RESET:
        .lword      start
;
```

图 2.24 sect30.inc 在单芯片模式中的列表 (4)

(e) 设置中断向量表

对于使用中断处理的程序，通过下列方法来设置中断向量表：
在 sect30.inc 中为向量段设置中断向量表。

中断向量的内容因不同的单片机类型而异，因此必须进行设置，以适合所使用的单片机类型。

有关详细资料，请参考单片机随附的用户手册。

(1) 当在 sect30.inc 中设置中断向量表时

对于使用中断处理的程序，在 sect30.inc 中更改向量段的中断向量表。

图 2.25 显示一个中断向量表的范例。

```
;------------------------------------------------------------------
；变量向量段
;------------------------------------------------------------------
        .section    vector,ROMDATA          ；变量向量表
        .org        VECTOR_ADR

        .lword      dummy_int               ； BRK （软件 int 0）
            :
        （已省略）
            :
        .lword      dummy_int               ； DMA0 （软件 int 8）
        .lword      dummy_int               ； DMA1 （软件 int 9）
        .lword      dummy_int               ； DMA2 （软件 int 10）
            :
        （已省略）
            :
        .lword      dummy_int               ； uart1 trance （软件 int 19）
        .lword      dummy_int               ； uart1 receive （软件 int 20）
        .lword      dummy_int               ； TIMER B0 （软件 int 21）
            :
        （已省略）
            :
        .lword      dummy_int               ； INT5 （软件 int 26）
        .lword      dummy_int               ； INT4 （软件 int 27）
            :
        （已省略）
            :
        .lword      dummy_int               ； uart2 trance/NACK （软件 int 33）
        .lword      dummy_int               ； uart2 receive/ACK （软件 int 34）
            :
        （已省略）
            :
        .lword      dummy_int               ；软件 int 63
```

* dummy_int 是一个虚设的中断处理函数。

图 2.25　　中断向量地址表

中断向量的内容因 M16C/60 系列和 R8C/Tiny 系列中的仿真器而异。参考仿真器的用户手册以获取详细资料。

更改中断向量地址表如下：

1. 在 .GLB as30 伪指令中进行中断处理函数的外部声明。NC30 所建立的函数标签的前缀将包含下划线 （_）。因此，在这里声明的中断处理函数的名称前缀也应该包含下划线。
2. 将中断处理函数的名称替换为使用 dummy_int 的中断处理函数名称，而此虚设中断函数名称（dummy_int）将与向量地址表中适当的中断表相对应。

图 2.26 是一个寄存 UART1 发送中断处理函数 uarttrn 的范例。

```
.lword    dummy_int        ;  uart0 receive （用于用户）
.glb      _uarttrn                                    ← 上面 (1) 的处理
.lword    _uarttrn         ;  uart1 trance （用于用户）  ← 上面 (2) 的处理

（已省略）
```

图 2.26　　设置中断向量地址的范例

3. 编程技术

本章描述在使用 C 编译器 NC30 进行编程时所须注意的事项。

3.1 注意

瑞萨科技的产品不是为了与人命相关的环境下所使用的设备或者系统而设计和制造的产品。在研讨将本资料所记载的产品用于运输、交通车辆、医疗、航天、原子能控制、海底中继器的设备或者系统等特殊用途时，请与瑞萨科技、瑞萨解决方案或者经授权的瑞萨半导体产品经销商联系。

3.1.1 有关编译器升级版的注意事项

由 NC30 生成的机器语言指令 （汇编语言）的内容，视编译时指定的启动选项、版本更改的内容等而有所不同。因此，在更改启动选项或升级编译器版本后，确保重新评价应用程序的操作。

此外，当在中断处理与非中断处理例程之间，或在实时 OS 中的任务之间引用相同的 RAM 数据 （并更改其内容）时，总是确保行使专属的控制，如 volatile 的指定。同时，也为具有不同成员名称，但却映射到相同的 RAM 的位字段结构行使专属的控制。

3.1.2 有关 M16C 的类型独有部分的注意事项

当在 SFR 区中写入或读取寄存器时，有时可能需要使用一个特定的指令。由于这个特定的指令依不同的单片机类型而异，请参考您单片机的用户手册以了解详细信息。在这里，使用 ASM 函数将指令直接写入到程序中。

在本编译器中，可能会生成无法使用的指令，写入及读取 SFR 区的寄存器。

当以 C 语言读取 SFR 区中的寄存器时，无论编译器是何版本及是否使用了优化选项，确保和使用 asm 函数时一样，生成相同的正确指令。

当您如下列范例所示的 C 语言描述，对 SFR 区进行描述时，由于中断请求位不正常，本编译器可能会生成多余且操作不良的汇编器代码。

```
#pragma  ADDRESS  TA0IC  006Ch                    /* M16C/60 MCU 的计时器 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 TA0IC.IR 变成 1 */
        {
            ;
        }
        TA0IC.IR = 0;                             /* 当它变成 1 时返回 0 到 TA0IC.IR */
}
```

图 3.1 对 SFR 区的 C 语言描述

3.1.3 关于优化

(a) 常用优化

无论是否指定优化选项，下列优化将总是执行。

(1) 无意义的变量存取

例如，下面显示的变量 port 并不使用读出结果，因此将删除读出操作。

```
extern int  port;

void    func(void)
{
        port;
}
```

图 3.2 无意义的变量存取的范例 （已优化）

虽然这个范例中所要进行的操作只是读出 port，但实际上读出代码在输出前并未被优化。若要禁止优化，可如图 3.3 所示添加 volatile 修饰符。

```
extern int volatile     port;

void    func(void)
{
        port;
}
```

图 3.3 无意义的变量存取的范例 （已禁止优化）

(2) 无意义的比较

```
int     func(char c)
{
        int             i;

        if(c != -1)
                        i = 1;
        else
                        i = 0;
        return i;
}
```

图 3.4 无意义的比较

在这个范例的情形中，变量 c 因为被设置为 char，所以编译器将它当作 unsigned char 类型。由于 unsigned char 类型所代表的值是 0 到 255，变量 c 将永远无法取得 -1 的值。

因此，若有任何语句像这个范例般在逻辑上是无效的，编译器将不会生成汇编器代码。

(3) 程序不被执行

对于逻辑上不被执行的程序，将不会生成任何汇编器代码。

```
void    func(int i)
{
        func2(i);
        return;

        i = 10;                                 ← 语句不被执行
}
```

图 3.5 程序不被执行

(4) 常数之间的运算

常数之间的运算在进行编译时被执行。

```
int     func(void)
{
        int             i = 1 + 2;              ← 这个部分的运算在进行编译时被执行

        return i;
}
```

图 3.6 程序不被执行

(5) 最优化指令的选择

无论是否指定优化选项，在使用 STZ 指令或为除法 / 乘法输出 shift 指令时，将总是执行最优化指令的选择。

(b) 关于 volatile 修饰符

volatile 修饰符的使用，可避免对变量的引用、它们被引用的顺序和次数等，受到优化的影响。

请避免编写如下所示意义含糊的语句。

```
int          a;
int volatile  b, c;

a = b = c;                    /* 到底 a = c 还是 a = b? */
a = ++b;                      /* 到底 a = b 还是 a = (b + 1)? */
```

图 3.7 具有意义含糊的 volatile 修饰符的范例

对于连续的位操作，若进行了优化，编译器将生成集体执行位操作的代码，即使在指定 volatile 修饰符后。（通过盖写引用的顺序，可同时进行位操作。）

若要禁止集体位操作，使用编译选项 “-Ono_bit （缩写 -ONB）”。

3.1.4 有关使用寄存器变量的注意事项

(a) 寄存器限定和编译选项 “-fenable_register(-fER)”

若指定编译选项 “-fenable_register(-fER)”，被限定寄存器以符合特定条件的变量，可被强制分配到寄存器。这项功能可在不依赖优化的情况下，改善所生成的代码。

由于这项功能若使用不当将产生负面影响，总是确保在使用前检查所生成的代码。

(b) 关于寄存器限定和优化选项

在指定优化选项后，作为一项优化功能，变量将被分配到寄存器。这项分配功能不受变量是否被限定寄存器影响。

3.1.5 关于启动处理

根据所使用的单片机类型或应用系统，可能需要对启动程序进行修改。对于与单片机的类型相关的修改，请参考单片机的数据手册等文档，然后在使用前更正编译器套件所随附的启动文件。

3.2 获取更高的代码效率

3.2.1 获取更高代码效率的编程技术

(a) 关于整数和变量

1. 除非有必要，否则使用无符号的整数。若 int、 short 或 long 类型没有符号说明符，它们将作为带符号的整数处理。除非有必要，为这些数据类型的整数运算添加 ‘unsigned’ 的符号说明符。[1]
2. 若可能，请勿使用 >= 或 ← 来比较带符号的变量。对条件判断使用 != 和 ==。

(b) far 类型数组

在机器语言级对 far 类型数组的引用因其大小而异。

1. 当数组大小在 64K 字节以内时
   下标以 16 位宽度计算。这将确保对大小为 64K 字节或以下的数组的有效存取。
2. 当数组大小大于 64K 字节或未知时
   下标以 32 位宽度计算。

因此，当知道数组的大小不超过 64K 字节时，如图 3.8 所示在 far 类型数组的 extern 声明中明确注明大小，或在编译前添加编译选项 “-fsmall_array(-fSA[2])”。这可帮助提高程序的代码效率。

```
extern int far    array[];     ← 大小未知，所以将大小计算为 32 位的值。
extern int far    array[10];   ← 大小在 64KB 内，所以存取更有效率。
```

图 3.8 far 数组的 extern 声明范例

1. 若 char 类型或位字段结构成员没有符号说明符，它们将被作为无符号处理。
2. 当指定 “-fsmall_array(-fSA)” 后，编译器在生成代码时，将假设具有未知大小的数组的大小是在 64K 字节以内。

(c) 有效的使用原型声明

NC30 允许通过声明函数的原型来完成有效的函数调用。

这表示除非函数在 NC30 中被声明其原型，否则当调用函数时，该函数的参数将按表 3.1 中所列出的规则保存在堆栈上。

表 3.1 对参数使用堆栈的规则

数据类型	保存在堆栈上的规则
char signed char	保存到堆栈时扩展到 int 类型。
float	保存到堆栈时扩展到 double 类型。
其它	保存到堆栈时不进行扩展。

基于这个原因，除非声明了函数的原型，否则 NC30 可能需要进行冗余的类型扩展。

函数的原型声明帮助禁止这类冗余的类型扩展，同时也使参数能够被分配到寄存器。所有这些操作可让您完成有效率的函数调用。

(d) 有效率使用 SB 寄存器

使用 SB 寄存器相关的寻址模式，将可以缩减应用程序的大小 （ROM 大小）。 NC30 可让您通过编写如图 3.9 所示的描述，来声明使用 SB 寄存器相关的寻址模式的变量。

```
#pragma SBDATA  val
int         val;
```

图 3.9 使用 SB 相关的寻址模式声明变量的范例

(e) 使用编译选项 -fJSRW 来压缩 ROM 大小

当在 NC30 中调用在文件以外定义的函数时，使用 JSR.A 指令来调用该函数。

然而，若程序不是太大，也可以使用 “JSR.W” 指令来调用大多数函数。

在这种情况下，可以通过下列操作来缩减 ROM 大小：

首先，使用 -fJSRW 选项来进行编译，然后检查在连接时被表示为错误的函数。接着仅将错误函数的声明更改为使用 “#pragma JSRA 函数名称” 的声明。

当使用 -OGJ 选项时，则在连接时为程序选用 JMP 指令。

(f) 其它方法

除了上述方法，也可以通过如下所示更改程序描述来压缩 ROM 容量。

1. 将只调用一次的相对较小的函数更改为直接插入函数。
2. 将 if-else 语句替换为 switch 语句。（这个做法十分有效，特别是针对数组、指针或结构等有关的非简单变量。）
3. 对于位比较，使用 ‘&’ 或 ‘|’ 来代替 ‘&&’ 或 ‘||’。
4. 对于仅返回 char 类型范围的值的函数，将返回值的类型声明为 char。
5. 对于与函数调用重叠的变量，请勿使用寄存器变量。

3.2.2 加速启动处理

ncrt0.a30 启动程序包含用于清除 bss 区的例程。如同 C 语言，这个例程确保未初始化的变量具有 0 的初始值。

例如，图 3.10 中显示的代码未初始化变量，因此该变量必须在启动例程中被初始化为 0（通过清除 bss[3] 区）。

```
static int          i;
```

图 3.10 没有初始值的变量的声明范例

在某些情况下，没有初始值的变量是不需要被清除至 0 的。在这种情形下，可以将启动程序中用于清除 bss 区的例程变成注释，以提高启动处理的速度。

```
;=====================================================================
; NEAR 区初始化。
;---------------------------------------------------------------------
; 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
                :
           （已省略）
                :
;=====================================================================
; FAR 区初始化。
;---------------------------------------------------------------------
; bss 零清除
;---------------------------------------------------------------------
;           BZERO       bss_SE_top,bss_SE
;           BZERO       bss_SO_top,bss_SO
```

图 3.11 将清除 bss 区的例程变成注释

[3] RAM 中没有初始值的外部变量用 “bss” 表示。

3.3 连接汇编语言程序与 C 程序

3.3.1 从 C 程序调用汇编函数

(a) 调用汇编函数

汇编函数和以 C 编写的函数一样，从 C 程序使用汇编函数的名称来进行调用。

汇编函数中的第一个标签前缀必须包含一个下划线 （_）。然而，当从 C 程序调用汇编函数时，该下划线将被省略。进行调用的 C 程序必须包含对汇编函数的原型声明。

图 3.12 是调用汇编函数 asm_func 的范例。

```
extern void     asm_func( void );                ← 汇编函数原型声明

void            main()
{
                    :
                （已省略）
                    :
                asm_func();                      ← 调用汇编函数
}
```

图 3.12 在不使用参数的情况下调用汇编函数的范例 （sample.c）

```
                .glb        _main
_main:
                    :
                （已省略）
                    :
                jsr         _asm_func            ← 调用汇编函数 （前缀包含 ‘_’）
                rts
```

图 3.13 sample.c 的编译结果 （sample.a30）

(b) 当分配参数到汇编函数时

将参数传递到汇编函数时，使用扩展函数 “#pragma PARAMETER”。这个 #pragma PARAMETER 会通过 32 位通用寄存器（R2R0、R3R1）、16 位通用寄存器（R0、R1、R2、R3）或 8 位通用寄存器（R0L、R0H、R1L、R1H），以及地址寄存器 （A0、 A1）将参数传递至汇编函数。

以下显示使用 #pragma PARAMETER 来调用汇编函数的操作顺序：

1. 在声明 #pragma PARAMETER 之前编写汇编函数的原型声明。同时也必须声明参数类型。
2. 在汇编函数的参数列表中通过 #pragma PARAMETER 声明所使用的寄存器的名称。

图 3.14 是在调用汇编函数 asm_func 时使用 #pragma PARAMETER 的一个范例。

```
extern unsigned int     asm_func(unsigned int, unsigned int);
#pragma PARAMETER  asm_func(R0, R1)                    ← 参数通过 R0 和 R1 寄存器
                                                         传递到汇编函数。
void       main(void)
{
           int             i = 0x02;
           int             j = 0x05;

           asm_func(i, j);
}
```

图 3.14 在使用参数的情况下调用汇编函数的范例 （sample2.c）

```
           .SECTION    program,CODE,ALIGN
           ._file      'sample2.c'
           .align
           ._line      5
;## # C_SRC :          {
           .glb        _main
_main:
           enter       #04H
           pushm       R1
           ._line      6
;## # C_SRC :                    int     i = 0x02;
           mov.w       #0002H,-4[FB]       ; i
           ._line      7
;## # C_SRC :                    int     j = 0x05;
           mov.w       #0005H,-2[FB]       ; j
           ._line      9
;## # C_SRC :                    asm_func(i, j);
           mov.w       -2[FB],R1   ; j              ← 参数通过 R0 和 R1 寄存器
           mov.w       -4[FB],R0   ; i                传递到汇编函数。
           jsr         _asm_func
           ._line      10
;## # C_SRC :          }
           popm        R1
           exitd
E1:
           .glb        _asm_func                ← 调用汇编函数 （前缀包含 ‘_’）
           .END                                 ← 通过 #pragma PARAMETER 指定的函数的输出汇编名称，
                                                前缀总是包含 _ （下划线）。
```

图 3.15 sample2.c 的编译结果 （sample2.a30）

(c) #pragma PARAMETER 声明中的参数的限制

下列参数类型无法在 #pragma PARAMETER 声明中进行声明。

- 结构类型和联合类型的参数
- 64 位整数类型 （long long）的参数
- 浮点类型 （double）的参数

另外，结构或联合类型的返回值也不可被定义为汇编函数的返回值。

3.3.2 编写汇编函数

(a) 编写调用目标汇编函数的方法

下面显示编写汇编函数的进入处理的步骤。

1. 使用汇编伪命令 .SECTION 来指定段的名称。
2. 使用汇编伪命令 .GLB 来全局指定函数名称的标签。
3. 将下划线 （_）添加到函数名称中以将它编写为标签。
4. 当修改函数中的 B 和 U 标志时，先将标志寄存器保存到堆栈。[4]

下面显示编写汇编函数的退出处理的步骤。

1. 如果在函数中修改了 B 和 U 标志，请从堆栈恢复标志寄存器。
2. 编写 RTS 指令。

请勿更改汇编函数中的 SB 和 FB 寄存器的内容。如果更改了 SB 和 FB 寄存器的内容，请在进入函数前将它们保存到堆栈，然后在退出函数后从堆栈恢复它们。

图 3.16 是如何编写汇编函数的范例。在此范例中，段的名称为 program，和 NC30 所输出的段的名称相同。

```
        .section    program                 ← (1)
        .glb        _asm_func               ← (2)
_asm_func:                                  ← (3)
        pushc       FLG                     ← (4)
        pushm       R3, R1                  ← (5)
        mov.w       SYM1, R1
        mov.w       SYM1+2, R3

        popm        R3, R1                  ← (6)
        popc        FLG                     ← (7)
        rts                                 ← (8)
        .END
```

图 3.16 编写汇编函数的范例

[4] 请勿更改汇编函数中的 B 和 U 标志的内容。

(b) 从汇编函数返回返回值

当从汇编函数返回值到 C 语言程序时，可使用寄存器来返回整数、指针和浮点类型的值。表 3.2 列出有关返回值的调用的规则。图 3.17 显示如何编写一个返回值的汇编函数的范例。

表 3.2 返回值的调用规则

返回值类型	规则
_Bool 类型 char 类型	R0L 寄存器
int 类型 near pointer 类型	R0 寄存器
float 类型 long 类型 far pointer 类型	返回值时， 16 低顺序位将存储在 R0 寄存器中，而 16 高顺序位则存储在 R2 寄存器中。
double 类型 long double 类型	返回值时，每个值以 16 位存储，从 MSB 开始，以 R3、 R2、 R1 和 R0 寄存器的顺序存储。
long long 类型	返回值时，每个值以 16 位存储，从 MSB 开始，以 R3、 R2、 R1 和 R0 寄存器的顺序存储。
复合类型	调用函数之前，存储区域的 far 地址返回值会被推进堆栈中。在返回进行调用的程序前，调用目标函数会将返回值写入推进堆栈中的 far 地址所表示的区域。

```
        .section    program
        .glb        _asm_func
_asm_func:
            :
        （已省略）
            :
        mov.w       #0A000H, R0
        mov.w       #0001H, R2
        rts
        .END
```

图 3.17 编写返回 long 类型返回值的汇编函数的范例

(c) 引用 C 变量

由于汇编函数是在和 C 程序不同的文件中编写，因此只有 C 全局变量可被引用。

当在汇编函数中包含 C 变量的名称时，请为它们的前缀加上一个下划线 （_）。另外，在汇编语言程序中，外部变量必须使用汇编伪指令 .GLB 来声明。

图 3.18 是从汇编函数 asm_func 引用 C 程序全局变量 counter 的范例。

```
C 程序：

unsigned int        counter;                    ← C 程序全局变量

void    main(void)
{
            :
        （已省略）
            :
}


汇编函数：

        .glb        _counter                    ← C 程序全局变量的外部声明
_asm_func:
            :
        （已省略）
            :
        mov.w       _counter, R0                ← 引用
```

图 3.18 引用 C 全局变量

(d) 有关在汇编函数中编写中断处理的注意事项

若您正在为中断处理编写程序（函数），必须在函数的进入和退出遵照下列处理来进行。

1. 在进入点保存寄存器（R0、R1、R2、R3、A0、A1 和 FB）。
2. 在退出点恢复寄存器（R0、R1、R2、R3、A0、A1 和 FB）。
3. 使用 REIT 指令从函数返回。

图 3.19 是为中断处理编写汇编函数的范例。

```
        .section    program
        .glb        _func
_int_func:
        pushm       R0,R1,R2,R3,A0,A1,FB        ← 保存寄存器
        mov.b       #01H, R0L
            :
        （已省略）
            :
        popm        R0,R1,R2,R3,A0,A1,FB        ← 还原寄存器
        reit                                    ← 返回到 C 程序
        .END
```

图 3.19 编写中断处理汇编函数的范例

(e) 有关从汇编函数调用 C 函数的注意事项

从汇编语言函数调用以 C 编写的函数时请注意下列事项。

1. 使用前缀包含了下划线 （_）或货币符号 （$）的标签来调用 C 函数。
2. 在调用 C 语言函数时， R0 寄存器及用于返回值的寄存器不会在 C 语言函数中保存。因此，当从汇编语言函数调用 C 语言函数时，在调用 C 语言函数前，先保存 R0 寄存器及用于返回值的寄存器。

3.3.3 有关编写汇编函数的注意事项

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) 有关处理 B 和 U 标志的注意事项

当从汇编函数返回到 C 语言程序时，总是确保 B 和 U 标志保持它们在函数被调用时的相同状态。

(b) 有关处理 FB 寄存器的注意事项

若您修改了汇编函数中的 FB （帧基址）寄存器，您可能无法从函数被调用的地方正常返回到 C 语言程序。

(c) 有关处理通用和地址寄存器的注意事项

通用寄存器 （R0、 R1、 R2、 R3）和地址寄存器 （A0、 A1）的内容可以在汇编函数中修改，而不会出现问题。

(d) 将参数传递到汇编函数

若需要将参数传递到以汇编语言编写的函数，可以使用 #pragma PARAMETER 函数。参数将会通过寄存器来传递。

图 3.20 中显示其格式 （图中的 asm_func 是一个汇编函数的名称）。

```
unsigned int near       asm_func(unsigned int, unsigned int);      ← 汇编函数的原型声明

#pragma PARAMETER asm_func(R0, R1)
```

图 3.20 汇编函数的原型声明

#pragma PARAMETER 通过 16 位通用寄存器 （R0、 R1、 R2、 R3）、 8 位通用寄存器 （R0L、 R0H、 R1L、 R1H），以及地址寄存器 （A0、 A1）将参数传递到汇编函数。此外， 16 位通用寄存器将结合地址寄存器，从而为传递至汇编函数的参数组成 32 位寄存器 （R3R1 和 R2R0）。请留意汇编函数的原型必须总是在声明 #pragma PARAMETER 之前进行声明。

然而，下列参数类型将无法在 #pragma PARAMETER 声明中声明：

- 结构或联合类型
- 64 位整数类型 （long long）的参数
- 浮点类型 （double）的参数

返回结构或联合类型的函数也无法被声明为函数的返回值。

3.4 其它

3.4.1 有关在 NC 系列编译器之间进行传输的注意事项

NC30 的语言指定基本上与瑞萨的 C 编译器 “NCxx”兼容 （包括扩展函数）。然而，如下面所述，本手册中的编译器与其它 NC 系列编译器之间有一些不同。

(a) 默认 near/far 的不同

NC 系列默认的 “near/far”在表 3.3 中显示。因此，当传输本手册中的编译器到其它 NC 系列编译器时，near/far 指定将需要做出调整。

表 3.3 NC 系列中的默认 near/far

编译器	RAM 数据	ROM 数据	程序
NC308	near （然而，指针类型是 far）	far	far 固定
NC30	near	far	far 固定
NC79	near	near	far
NC77	near	near	far

3.4.2 有关在 NC308 和 NC30 之间进行传输的注意事项

(a) 调用惯例的不同

在 NC30 中，调用函数时保存寄存器的操作是在调用函数中执行，但在 NC308 中，这项操作则是在调用目标函数中 （主体）执行。基于这个原因，若要让 NC30 中的 C 函数调用汇编函数，请执行下列步骤：

条件：
当汇编函数中寄存器的值被破坏时，
(1) 在调用汇编函数前保存寄存器
(2) 在从汇编函数返回后返回寄存器

附录 A　命令选项参考

本附录将描述如何启动编译驱动器 nc30 及有关的命令行选项。对命令行选项的描述中，包括了可从 nc30 启动的 as30 汇编器和 ln30 连接编辑器。

附录 A.1　　nc30 命令格式

```
% nc30Δ[ 命令行选项 ]Δ[ 汇编语言源文件名称 ]Δ[ 可再定位的模块文件名称 ]Δ<C 源文件名称 >

%　　：　提示符
< >　：　强制性项目
[ ]　：　可选项目
Δ　　：　空格
```

附图 A.1　　nc30 命令行格式

```
% nc30  -osample  -as30  "-l"  -ln30  "-ms"  ncrt0.a30  sample.c<RET>

<RET>　：　回车键
* 连接时总是先指定启动程序。
```

附图 A.2　　nc30 命令行的范例

附录 A.2 nc30 命令行选项

附录 A.2.1 用于控制编译驱动器的选项

附表 A.1 显示用于控制编译驱动器的命令行选项。

附表 A.1 用于控制编译驱动器的选项

选项	功能
-c	建立可再定位的模块文件 （扩展名 .r30）并结束处理 [1]。
- D *标识符*	定义标识符。与 #define 功能相同。
-dsource （缩写 -dS）	生成汇编语言源文件，并将其中的 C 语言源列表输出变为注释 （扩展名 “.a30”）。（汇编后也不会将其删除。）
-dsource_in_list （缩写 -dSL）	除 “-dsource(-dS)” 的功能外，可生成汇编语言列表文件 （.lst）。
-E	仅调用预处理命令，并将结果输出为标准输出。
-I *目录*	指定包含在 #include 中指定的文件的目录。 您最多可以指定 50 个目录。
-P	仅调用预处理命令并建立文件 （扩展名 .i）。
-S	建立汇编语言源文件 （扩展名 .a30）并结束处理。
-silent	禁止启动时的版权信息显示。
-U *预定义的宏*	取消指定的预定义宏。

-c

编译驱动器控制

功能： 建立可再定位模块文件 （扩展名 .r30）并完成处理。

注意： 如果指定了此选项，则不会建立由 ln30 输出的绝对模块文件 （扩展名 .x30）或其它文件。

- D *标识符*

编译驱动器控制

功能： 此功能与预处理命令 #define 相同。
以空格分隔多个标识符。

语法： nc30Δ-D *标识符* [= *常数*]Δ<C 源文件 >

[= *常数*] 是可选的。

注意： 可以定义的标识符数取决于主机操作系统的命令行所允许指定的最大字符数。

1. 若未指定 -c、-E、-P 或 -S 的命令行选项， nc30 将在 ln30 完成处理，并输出已建立的文件，包括绝对加载模块文件 （扩展名为 .x30）在内。

-dsource -dS

注释选项

功能： 生成汇编语言源文件，并将其中的 C 语言源列表输出变为注释 （扩展名 “.a30”）。（汇编后也不会将其删除）。

补充：
1. 使用 -S 选项即自动允许 “-dsource(-dS)” 选项。
2. 生成文件 “.a30” 和 “.r30” 将不会被删除。如果您需要将 C 语言源列表输出至汇编列表文件时，请使用此选项。

-dsource_in_list -dSL

列表文件选项

功能： 除 “-dsource(-dS)” 的功能外，可生成汇编语言列表文件 （文件扩展名 “.lst”）。

-E

编译驱动器控制

功能： 仅调用预处理命令并将结果输出到标准输出。

注意： 如果指定此选项，则不会生成由 ccom30、 as30 或 ln30 输出的汇编源文件 （扩展名 .a30）、可再定位模块文件 （扩展名 .r30）、绝对模块文件 （扩展名 .x30）或其它文件。

-I *目录*

编译驱动器控制

功能： 指定目录名，然后在其中搜索将由预处理命令 #include 引用的文件。
最多指定 50 个目录。

语法： nc30Δ-I *目录*Δ<C 源文件 >

注意： 可以定义的目录数取决于主机操作系统的命令行所允许指定的最大字符数。

-P

编译驱动器控制

功能： 仅调用预处理命令，建立文件 （扩展名 .i）并停止处理。

注意：
1. 如果指定此选项，则不会生成由 ccom30、 as30 或 ln30 输出的汇编源文件 （扩展名 .a30）、可再定位模块文件 （扩展名 .r30）、绝对模块文件 （扩展名 .x30）或其它文件。
2. 由此选项生成的文件 （扩展名 .i）不包含由预处理器生成的 #line 命令。要获得包含 #line 的结果，请使用 -E 选项。

-S

编译驱动器控制

功能： 建立汇编语言源文件 （扩展名 .a30 和 .ext）并停止处理。

注意： 如果指定此选项，则不会生成由 as30 或 ln30 输出的可再定位模块文件 （扩展名 .r30）、绝对模块文件 （扩展名 .x30）或其它文件。

-silent

编译驱动器控制

功能： 禁止在启动时显示版权通知。

-U *预定义的宏*

编译驱动器控制

功能： 对预定义的宏常数取消其定义。

语法： nc30Δ-U *预定义的宏*Δ<C 源文件 >

注意： 允许取消其定义的最大宏数取决于主机操作系统的命令行所允许指定的最大字符数。
STDC、 _LINE_、 _FILE_、 _DATE_ 和 _TIME_ 不能取消定义。

附录 A.2.2 指定输出文件的选项

附表 A.2 显示指定输出机器语言数据文件的名称的命令行选项。

附表 A.2 指定输出文件的选项

选项	功能
-dir *目录名称*	指定 ln30 生成的文件 （绝对模块文件、映像文件等）的目标目录。
-o *文件名*	指定 ln30 生成的文件 （绝对模块文件、映像文件等）的名称。此选项也可用于指定目标目录。 此选项同时可用于指定文件名，包括路径。不要指定文件扩展名。

-dir *目录名称*

输出文件的指定

功能： 此选项允许您为输出文件指定输出的目标目录。

语法： nc30Δ-dir *目录名称*

注意： 用于调试的源文件信息将从调用编译器的目录 （当前目录）开始生成。
所以，如果输出文件在不同的目录中生成，则必须向调试器等通知调用编译器的目录。

-o *文件名*

输出文件的指定

功能： 指定 ln30 生成的文件 （绝对模块文件、映射文件等）的名称。此选项同时可用于指定文件名，包括路径。
不可指定文件扩展名。

语法： nc30Δ-o *文件名* Δ<C 源文件 >

附录 A.2.3 版本信息显示选项

附表 A.3 显示用于显示交叉工具版本数据的命令行选项。

附表 A.3 显示版本数据的选项

选项	功能
-v	在执行期间显示命令程序的名称和命令行。
-V	显示编译器的启动信息，然后完成处理 （不编译）。

-v

显示命令程序名称

功能： 在编译文件的同时显示正在执行的命令程序名称。

注意： 对于此选项，请使用小写 v。

-V

显示版本数据

功能： 显示由编译器执行的命令程序的版本数据，然后完成处理。

补充： 使用此选项以检查是否正确安装编译器。“M16C Family C Compiler package Release Notes”（M16C 族 C 编译器套件发行说明）列出了由编译器在内部执行的命令的正确版本号。

如果发行说明中列出的版本号与使用此选项所显示的版本号不符，则您有可能未正确安装套件。有关安装 NC30 套件的详情，请参考 “M16C Family C Compiler package Release Notes”（M16C 族 C 编译器套件发行说明）。

注意：
1. 对于此选项，请使用大写 V。
2. 如果指定此选项，将忽略所有其它选项。

附录 A.2.4 用于调试的选项

附表 A.4 显示用于为 C 源文件输出符号文件的命令行选项。

附表 A.4 用于调试的选项

选项	功能
-g	将调试信息输出到一个汇编源文件 （扩展名为 .a30）。因此，能够执行 C 语言级的调试。
-genter	调用函数时，总是输出 enter 指令。 在使用调试器的堆栈跟踪功能时，确保指定此选项。
-gno_reg	禁止输出寄存器变量的调试信息。
-gold	此选项将以 Rev.E 格式输出调试信息。 当指定此选项后，将自动指定 “-gno_reg”选项和 “-fauto_128”选项。

-g

输出调试信息

功能： 将调试信息输出至汇编源文件 （扩展名 .a30）。

注意： 在 C 语言级调试程序时，请总是指定此选项。此选项的指定不影响编译器的代码生成。
当指定 “-finfo”选项后，此选项将有效。

-genter

输出 enter 指令

功能： 调用函数时，总是输出 enter 指令。

注意：
1. 在使用调试器堆栈跟踪功能时，总是指定此选项。如果不指定此选项，您将无法得到正确的结果。
2. 当指定此选项后，无论是否必要，编译器都将在进入函数时使用 enter 命令生成代码，以重建堆栈帧。因此， ROM 大小和使用的堆栈数将会增加。

-gno_reg

禁止有关寄存器变量的调试信息

功能： 禁止输出寄存器变量的调试信息。

补充： 如果不需要有关寄存器变量的调试信息，请使用此选项禁止输出此信息。禁止输出有关寄存器变量的调试信息将加快下载到调试器的速度。

-gold

以先前格式输出调试信息

功能： 此选项将以 Rev.E 格式输出调试信息。
当指定此选项后，将自动指定 “-gno_reg”选项和 “-fauto_128”选项。

补充： 从 NC30 V.2.00 开始，随着 auto 变量最大数的提高，调试信息的格式已更改 （从 xxx.r30 和 xxx.x30 格式起）。新的格式称为 Rev. F 格式。新格式 （xxx.x30）的可执行目标文件与下列调试器兼容：

1. PDB30 V.2.00 及更高版本
2. PDB30SIM V.2.00 及更高版本
3. HEW V.4.00 及更高版本

如果您正在使用的调试器不能以新格式 （xxx.x30）加载可执行目标文件，请在编译时使用 -gold 选项。

附录 A.2.5 优化选项

附表 A.5 显示用于优化程序执行速度和 ROM 容量的命令行选项。

附表 A.5 优化选项

选项	缩写	功能
-O[1-5]	无	优化速度和 ROM 大小。
-OR	无	优化 ROM 大小。
-OS	无	优化速度。
-OR_MAX	-ORM	执行 ROM 大小优先的优化。
-OS_MAX	-OSM	执行速度优先的优化。
-Ocompare_byte_to_word	-OCBTW	在相邻的地址上，以字的格式来比较字节数据。
-Oconst	-OC	通过常数替换 （在 const 修饰符的声明中）外部变量的引用来执行优化。
-Ofloat_to_inline	-OFTI	扩展直接插入的浮点运行时程序库，以加速浮点运算的处理。 （仅适用于比较和乘法）
-Oforward_function_to_inline	-OFFTI	扩展所有的直接插入函数。
-Oglb_jmp	-OGJ	优化全局跳转。
-Oloop_unroll[= *循环计数*]	-OLU	按照循环计数的次数展开代码，而无须让循环语句循环出现。“循环计数”可以省略。省略 “循环计数”后，此选项将应用最多 5 次的循环计数。
-Ono_asmopt	-ONA	禁止启动汇编优化器 “aopt30”。
-Ono_bit	-ONB	根据位操作的组合来禁止优化。
-Ono_break_source_debug	-ONBSD	禁止会影响源数据的优化。
-Ono_float_const_fold	-ONFCF	禁止浮点数的常数合并处理。
-Ono_logical_or_combine	-ONLOC	禁止将连续的 OR 放在一起的优化。
-Ono_stdlib	-ONS	禁止标准程序库函数的直接插入填充和程序库函数的修改。
-Osp_adjust	-OSA	优化堆栈校正码的删除。这可减少 ROM 所需的容量。 然而，这可能导致所使用的堆栈数增加。请使用 -O[1-5] 来指定此选项。
-Ostack_frame_align	-OSFA	在偶数边界对齐堆栈帧。
-Ostatic_to_inline	-OSTI	静态函数被当作直接插入函数处理。
-O5OA	无	在选择优化选项 “-O5”后，禁止根据位操作指令生成代码。

主要优化选项的效果在附表 A.6 中显示。

附表 A.6 各项优化选项的效果

选项	-O	-OR	-OS	-OSA	-OSFA
SPEED	更快	更低	更快	更快	更快
ROM 大小	缩减	缩减	增加	缩减	相同 [2]
堆栈的使用	缩减	相同	相同	增加	增加

2. -OSFA 将每个进入函数的堆栈地址调整为偶数地址。在大多数的函数没有堆栈帧的情况下，程序的大小将会增加。

-O[1-5]

优化

功能： 优化速度和 ROM 大小。此选项可使用 -g 选项来指定。若您未指定数值 （无级别），则将假设为 -O3。

-O1： 此选项所执行的一些具代表性的优化项目包括下列各项。
- 将变量分配到寄存器。
- 删除无意义的条件表达式。
- 删除未按逻辑执行的语句。

-O2： 与 “-O1”无区别。

-O3： 执行 “-O1”以外的一些优化项目。
此选项所执行的一些具代表性的优化项目包括下列各项。
- 位操作分组。
- 浮点数字的常数合并处理。
- 标准程序库函数的直接插入填充。

-O4： 执行 “-O3”以外的一些优化项目。
此选项所执行的一些具代表性的优化项目包括下列各项。
- 将在 const 修饰符中声明的变量的引用替换为常数。

-O5： 执行 “-O4”以外的一些优化项目。
此选项所执行的一些具代表性的优化项目包括下列各项。
- 指针和结构等地址计算的优化 （若同时指定了 “-OR”选项）。
- 加强指针的优化 （若同时指定了 “-OS”选项）。

但是，当符合以下条件时，则可能无法输出正常代码。
- 在一个函数中有多个变量同时指定相同的存储器位置。

范例：

```
int     a = 3;
int     *p = &a;

void    test1(void)
{
        int             b;
        *p = 9;
        a = 10;
        b = *p;                 /* 通过应用优化 , "p" 将被转置为 "9". */
        printf( "b = %d (expect b = 10)\n",b );
}
```

结果：

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

-O[1-5]

优化

注意：

在使用了 “-O5”优化选项后，某些情况下，编译器将生成 “BTSTC”或 “BTSTS”位操作指令。在 M16C/60 对其 SFR 区中的存储器进行读写时，将不能够使用如 BTSTC、BTSTS 的位操作指令。

但是，编译器并不会辨认任何寄存器的类型，因此，如果为中断控制寄存器生成了 “BTSTC”或 “BTSTS”指令，则汇编器将与您所要开发的程序有所不同。

当在程序中使用了下面所示的 “-O5”优化选项后，“BTSTC”指令将在编译时生成，同时会阻止中断请求位的正确处理，导致汇编器执行错误的操作。

在编译时不可使用优化选项的 C 源：

```
#pragma  ADDRESS  TA0IC  006Ch  /* M16C/60 MCU 的计时器 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 TA0IC.IR 变成 1 */
        {
                ;
        }
        TA0IC.IR = 0;                       /* 当它变成 1 时返回 0 到 TA0IC.IR */
}
```

如果位操作指令被生成到 SFR 区，则请在采取以下措施后进行编译。确保在即使生成 “BTSTC”和 “BTSTS”指令的情况下，也不会带来不良后果。

- 使用除了 “-O5”以外的优化选项。
- 指令将在程序中使用 ASM 函数来直接描述。

-OR

优化

功能： ROM 大小的优化比速度优先。可以使用 “-g”和 “-O”选项来指定此选项。

注意： 使用此选项时，部分的源信息会在优化过程中被修改。因此，如果指定了此选项，则在调试器中运行您的程序时，程序有可能执行不同的操作。
如果您不希望源信息被修改，则使用 “-One_break_source_debug(-ONBSD)”选项来禁止优化。

-OS

优化

功能： 尽管 ROM 大小可能会有所增加，但执行优化可以将速度提高到最快。
此选项可以与 “-g”选项一同指定。

-OR_MAX -ORM

优化

功能： 执行 ROM 大小优先的优化。

解释：

1. 下面所列的编译选项将启用。
   - -O5
   - -OR
   - -O5OA
   - -Oglb_jmp (-OGJ)
   - -fchar_enumerator (-fCE)
   - -fdouble_32 (-fD32)
   - -fno_align (-fNA)
   - -fno_carry (-fNC)
   - -fsmall_array (-fSA)
   - -fuse_DIV (-fUD)
2. 若在综合开发环境或 HEW 中使用此选项，请确保在瑞萨 M16C 标准工具链的 C 标签上启用 “Size or speed:”（大小或速度：），然后选择 “ROM size to the minimum”（ROM 大小缩减至最小）。

注意：

1. 某些情况下，编译器会生成 “BTSTC”或 “BTSTS”的位操作指令。在 M16C 族中，“BTSTC”和 “BTSTS”位操作指令被禁止重写中断控制寄存器的内容。
   但是，编译器并不会辨认任何寄存器的类型，因此，如果为中断控制寄存器生成了包含 “BTSTC”或 “BTSTS”指令的汇编程序，则此汇编程序将与您在某一特定单片机上所要开发的程序有所不同。
   如果位操作指令被生成到 SFR 区，则请在采取以下措施后进行编译。确保在即使生成 “BTSTC”和 “BTSTS”指令的情况下，也不会带来不良后果。
   - 选择除了 “-OR_MAX”或 “-O5”以外的编译选项。
   - 指令将在程序中使用 ASM 函数来直接描述。
2. 源信息的一部分可在优化过程中修改。因此，如果指定了此选项，则在调试器中运行您的程序时，程序有可能执行不同的操作。如果您不希望源信息被修改，则使用 “-One_break_source_debug(-ONBSD)”编译选项来禁止优化。
3. 请确保指定连接选项 “-JOPT”。
4. 在一些调试器中， enum 类型可能不会被正确引用。
5. 必须总是明确编写函数的原型。若没有原型声明，编译器可能无法生成正确的代码。
6. double 类型的调试信息将作为 float 类型处理。因此， double 类型的数据会在 C 监视窗口及 Debug （调试）工具的全局窗口中显示为 float 类型。
7. 当 far 类型的指针用于间接存取通过 malloc 等函数动态分配的存储器或映射至 far 区域的 ROM 数据时，请确保存取的数据未超出 64K 字节的边界。
8. 此选项不能与 “-R8C”选项同时使用。此选项不能与 “-R8C”选项同时使用。
9. 如果除法运算导致溢出，则编译器的运行可能与 ANSI 中的规定有所不同。

-OS_MAX -OSM

优化

功能： 执行速度优先的优化。

解释：

1. 下面所列的编译选项将启用。
   - -O4
   - -OS
   - -Oglb_jmp (-OGJ)
   - -Oloop_unroll=10 (-OLU=10)
   - -Ostatic_to_inline (-OSTI)
   - -fchar_enumerator (-fCE)
   - -fdouble_32 (-fD32)
   - -fno_carry (-fNC)
   - -fsmall_array (-fSA)
   - -fuse_DIV (-fUD)
2. 若在综合开发环境或 HEW 中使用此选项，请确保在瑞萨 M16C 标准工具链的 C 标签上启用 “Size or speed:”（大小或速度：），然后选择 “ROM size to the minimum”（ROM 大小缩减至最小）。

注意：

1. 请确保指定连接选项 “-JOPT”。
2. 由于 “for”语句循环出现，所以 ROM 大小会有所增加。
3. 系统将总是生成用于静态函数 （已被作为直接插入函数处理）功能描述的汇编器代码。
4. 如果要将直接插入函数强制到程序里，必须做出内部声明。
5. 在一些调试器中， enum 类型可能不会被正确引用。
6. 必须总是明确编写函数的原型。若没有原型声明，编译器可能无法生成正确的代码。
7. double 类型的调试信息将作为 float 类型处理。因此， double 类型的数据会在 C 监视窗口及 Debug （调试）工具的全局窗口中显示为 float 类型。
8. 当 far 类型的指针用于间接存取通过 malloc 等函数动态分配的存储器或映射至 far 区域的 ROM 数据时，请确保存取的数据未超出 64K 字节的边界。
9. 如果此选项与 “-R8C”选项同时使用时， “-fno_carry(-fNC)”选项的作用将变为无效。
10. 如果除法运算导致溢出，则编译器的运行可能与 ANSI 中的规定有所不同。

-Ocompare_byte_to_word -OCBTW

优化

功能： 在相邻的地址上，以字的格式来比较字节数据。

-Oconst -OC

优化

功能： 通过将 const 修饰符所声明的变量引用替换为常数来优化代码生成。
这在指定了 “-O4”以外的选项时也有效。

补充： 当符合下列所有条件时，将执行优化：
1. 变量不包括位字段与联合。
2. 变量指定了 const 修饰符，但未指定 volatile。
3. 变量将在相同的 C 语言源文件中进行初始化。
4. 变量由常数或限定为 const 的变量进行初始化。

-Ofloat_to_inline -OFTI

优化

功能： 扩展直接插入的浮点运行时程序库，以提高浮点算术运算的速度 （仅限于比较和乘法）。

-Oforward_function_to_inline -OFFTI

优化

功能： 扩展所有的直接插入函数。

补充： 虽然直接插入函数需要在进行函数定义后才能被调用，不过使用此选项将可在调用直接插入函数后才进行函数定义。

注意：
1. 当为函数指定直接插入存储类时，确保将直接插入存储类及其程序定义编写到与该函数相同的文件中。
2. “结构”和 “联合”不能使用直接插入函数的参数，否则将出现编译错误。
3. 无法对直接插入函数进行间接调用。一旦以间接调用来编程，就会发生编译错误。
4. 无法对直接插入函数进行递归调用。一旦以递归调用来编程，就会发生编译错误。

-Oglb_jmp -OGJ

优化

功能： 优化全局跳转。

注意： 当使用此选项时，请确保指定连接选项 “JOPT”。

-Oloop_unroll[= *循环计数*] -OLU[= *循环计数*]

展开循环

功能： 按照循环计数的次数展开代码，而无须重复执行循环语句。
“循环计数”可以省略。省略 “循环计数”后，此选项将应用最多 5 次的循环计数。

补充： 唯有从 “for”语句获得已知的执行次数时才展开代码。
默认情况下，此选项将被应用至最多循环出现五次 “for”的 “for”语句。

注意： 由于 “for”语句循环出现的原因，所以 ROM 大小会有所增加。

-Ono_asmopt -ONA

禁止启动汇编优化器

功能： 禁止启动汇编优化器 “aopt30”。

-Ono_bit -ONB

禁止优化

功能： 根据位操作的分组来禁止优化。

补充： 如果您指定了 -O[3-5]（或 -OR 或 -OS），则为分布在相同存储区里的位字段，整合常数替换的连续操作，以单一操作进行优化。
当连续位操作的顺序存在时，如在 I/O 位字段中，就不适宜执行此操作。因此，可使用本选项来禁止优化。

注意： 1. 将执行此优化，并忽略 volatile 的修饰来指定变量。
2. 只有在您指定了选项 “-O[3-5]”（或 “-OR”或 “-OS”）时，此选项才有效。

-Ono_break_source_debug -ONBSD

禁止优化

功能： 禁止会影响源数据的优化。

补充： 指定 “-OR”或 “-O”选项将执行以下优化，有可能会影响源数据。此选项（“-ONBSD”）用于禁止这类优化。

注意： 只有在指定 “-OR”或 “-O”选项后，此选项才有效。

-Ono_float_const_fold -ONFCF

禁止优化

功能： 禁止浮点数的常数合并处理。

补充： 默认情况下， NC30 会合并常数。以下是一个范例。

优化前：
(val/1000e250)*50.0

优化后：
val/20e250

在这种情况下，如果应用程序使用浮点的全动态范围，则计算结果将随计算顺序的更改而有所不同。此选项将禁止浮点数字的常数合并，因此 C 源文件中的计算顺序得以保留。

-Ono_logical_or_combine -ONLOC

禁止优化

功能： 禁止将连续的 OR 放置在一起的优化。

补充： 若按照以下范例所示，在进行编译时指定了 “-O3 or greater （-O3 或以上）、 -OR 或 -OS” 这三个选项之一，编译器将通过合并逻辑 OR 来优化代码生成。

范例：

```
if( a & 0x01 ¦¦ a & 0x02 ¦¦ a & 0x04 )
                ↓   ( 已优化 )
if ( a & 0x07)
```

在这种情况下，变量 a 将被引用多达三次，但在优化后，它将只被引用一次。
但是，如果变量 a 对 I/O 引用等有任何影响，则程序可能会因为优化而无法正确运行。
在这种情况下，指定此选项将禁止组合逻辑 OR 的优化。不过，如果以 volatile 声明变量，则不会组合 OR 以进行优化。

-Ono_stdlib -ONS

禁止优化

功能： 禁止标准程序库函数的直接插入填充、程序库函数的修改和其它类似的优化处理。

补充： 此选项将禁止以下优化。

- 对用 SMOVF 指令等替换如 “strcpy()” 和 “memcpy()” 的标准程序库函数的优化。
- 更改为遵循 near 和 far 参数的程序库函数的优化。

注意： 当编写的函数名称与标准程序库函数的名称相同时，指定此选项。

-Osp_adjust -OSA

调用函数后删除堆栈校正代码

功能： 通过在调用函数后合并堆栈校正代码来优化代码生成。

补充： 由于通常在每次调用函数后，都将取消为函数分配的参数区，所以将执行校正堆栈指针的处理。
若指定了此选项，则校正堆栈指针的处理将集中执行，而不是为每次函数调用分别执行。

范例：

在下面所示的范例中，堆栈指针在每次调用 func1() 和 func2() 时被校正，因此堆栈指针会被校正两次。若指定了此选项，堆栈指针将只会校正一次。

```
long    func1(long, long);
long    func2(long);

void    main( void )  {
            long          i = 1;
            long          j = 2;
            long          k,n;

            k = func1( i, j );
            n = func2( k );
}
```

注意： 使用选项 “-Osp_adjust” 有助于减小所需的 ROM 容量，同时加快处理速度。但是，使用的堆栈数将可能增加。
请在指定此选项时，也一同指定 -O[1-5]、 -OR、 -OS 选项。

-Ostack_frame_align -OSFA

对齐堆栈帧

功能： 在偶数边界对齐堆栈帧。
在评价版本中，将不能指定此选项。

补充： 当偶数大小的 auto 变量被映射至奇数地址时，对存储器的存取将比变量被映射至偶数地址多出一个周期。此选项将偶数大小的 auto 变量映射至偶数地址，从而加快了对存储器的存取速度。

注意：
1. 下面用 #pragma 指定的函数将不被对齐。
   - #pragma INTHANDLER
   - #pragma HANDLER
   - #pragma ALMHANDLER
   - #pragma CYCHANDLER
   - #pragma INTERRUPT[3]
2. 确保在启动程序将堆栈点初始化为偶数地址。另外，也确保使用此选项来编译所有程序。

[3] 由于无法保证在中断时生成的堆栈指针的值是偶数，因此对中断函数不进行对齐。基于这个原因，当对从中断函数调用的函数指定“-Ostack_frame_align”选项时，处理速度可能变慢。

-Ostatic_to_inline -OSTI

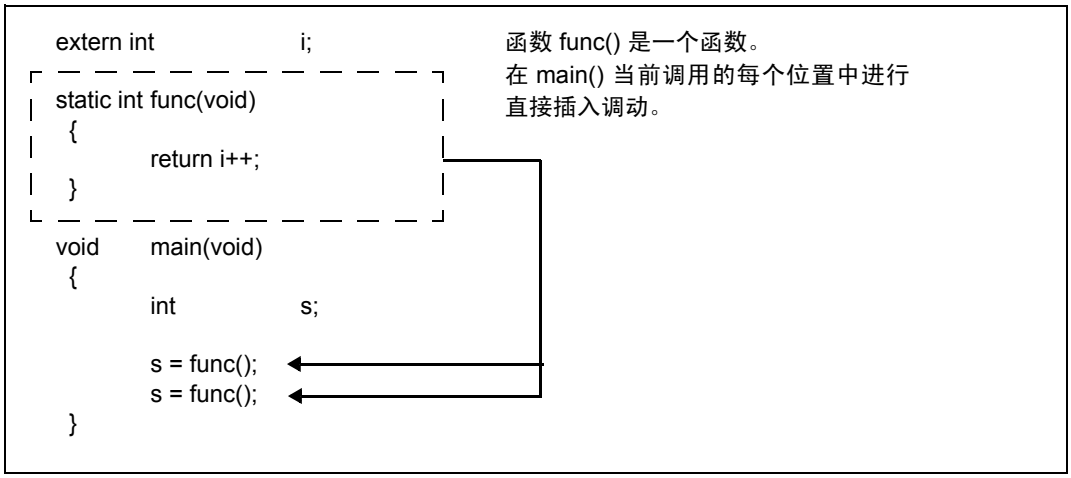
把 static 函数当作直接插入函数

功能： 将 static 函数视为直接插入函数，并执行已生成且直接插入的汇编代码。

补充： 满足以下条件时， static 函数将被视为直接插入函数，并执行已生成且直接插入的汇编代码。

1. 针对 static 函数，在调用函数前，必须描述其主体程序。
   - 函数调用和该函数的主体必须在同一个源文件中编写。
   - 当您指定 “-Oforward_function_to_inline”选项时，可忽略这个条件。
2. 在程序中省略了对 static 函数的地址获取。
3. 未执行 static 函数的递归调用。
4. 未在编译器的汇编代码输出中执行帧的建设 （如提供 auto 变量的保存等）。
   - 存在帧建设的情况下，随着目标函数描述内容与另一优化选项的结合使用而更改。
   - 当您指定 “-Oforward_function_to_inline”选项时，可忽略这个条件。

下面是直接插入调动的执行。其中显示了描述 static 函数的范例。



注意：

1. 总是生成汇编器代码，此代码描述被当作直接插入函数的 static 函数。
2. 请针对强制为直接插入的函数， 作出直接插入声明。

-O5OA

禁止代码生成

功能： 在选择优化选项 “-O5”后，禁止根据位操作指令 （BTSTC 和 BTSTS）生成代码。

注意： 位操作指令 （BTSTC 和 BTSTS）不可用来读取或写入 SFR 区内的寄存器。在选定了 “-O5”优化选项时选择此选项，将使用位操作指令生成用于读取或写入 SFR 区内的寄存器的代码。

附录 A.2.6 修改生成的代码的选项

附表 A.7 到附表 A.8 显示用于控制 nc30 所生成的汇编代码的命令行选项。

附表 A.7 修改生成的代码的选项 (1)

选项	缩写	功能
-fansi	无	使 “-fnot_reserve_far_and_near”、“-fnot_reserve_asm”和 “-fextend_to_int”有效。
-fchar_enumerator	-fCE	将 enumerator 类型作为 unsigned char 类型处理，而不是作为 int 类型处理。
-fconst_not_ROM	-fCNR	不会将 const 指定的类型作为 ROM 数据处理。
-fdouble_32	-fD32	此选项指定将 double 类型像 float 类型一样，以 32 位的数据长度处理。
-fenable_register	-fER	使寄存器存储类可用。
-fextend_to_int	-fETI	将 char 类型的数据扩展为 int 类型后执行操作。（根据 ANSI 的标准来扩展。）[4]
-ffar_RAM	-fFRAM	将 RAM 数据的默认属性更改为 far。
-finfo	无	将 “STK Viewer”、“Map Viewer”和 “utl30”所需的信息输出到绝对模块文件 （.x30）。
-fJSRW	无	将调用函数的默认指令更改为 JSR.W。当指定 -OGJ 时，则不需要执行此选项。
-fbit	-fB	生成代码并基于假设可以通过使用绝对寻址，对映射到 near 区域的所有外部变量按位操作指令执行。
-fno_carry	-fno_carry	当使用 far 类型指针间接存取数据时，禁止进位标志的加法。
-fauto_128	-fA1	将可用的堆栈帧限制为 128 字节。
-ffar_pointer	-fFP	将 pointer 类型变量的默认属性更改为 far。
-fnear_ROM	-fNROM	将 ROM 数据的默认属性更改为 near。
-fno_align	-fNA	不对齐函数的开始地址。
-fno_even	-fNE	将所有数据分配到奇数段，输出时不分开奇数数据与偶数数据。
-fno_switch_table	-fNST	指定此选项时，在比较之后发生转移的代码将生成到一个 switch 语句中。
-fnot_address_volatile	-fNAV	不将 #pragma ADDRESS （#pragma EQU）指定的变量视为 volatile 指定的变量。
-fnot_reserve_asm	-fNRA	保留字不包括 asm。（只有 _asm 有效。）
-fnot_reserve_far_and_near	-fNRFAN	保留字不包括 far 和 near。（只有 _far 和 _near 有效。）
-fnot_reserve_inline	-fNRI	保留字不包括 far 和 near。（只有 _inline 作为保留字。）
-fsmall_array	-fSA	引用 far 类型的数组时，如果在编译时不知其总大小，此选项将假设数组总大小低于 64K 字节，并以 16 位计算下标。
-fswitch_other_section	-fSOS	此选项将 ‘switch’语句的 ROM 表输出至其它段，而不是程序段。
-fchange_bank_always	-fCBA	此选项允许您将多个变量写入扩展区。
-fauto_over_255	-fAO2	将每个函数可保留的堆栈帧大小更改为 64K 字节。

4. 在 ANSI 规则下求值所得的 char 类型数据或 signed char 类型数据，将总是被扩展为 int 类型数据。
这是因为不这么做的话， char 类型的运算 （例如 c1=c2*2/c3）将发生溢出，并无法获取所要的结果。

附表 A.8　修改生成的代码的选项 (2)

选项	缩写	功能
-fsizet_16	-fS16	将 size_t 的类型定义从 unsigned long 类型更改为 unsigned int 类型。
-fptrdifft_16	-fP16	将 ptrdiff_t 的类型定义从 signed long 类型更改为 signed int 类型。
-fuse_DIV	-fUD	此选项将更改除法运算的生成代码。
-fuse_MUL	-fUM	此选项将更改乘法运算的生成代码。
-fSB_auto	-fSBA	在生成 SB 相对寻址前，一次将一个函数从一个 SB 寄存器更换到另一个 SB 寄存器。
-R8C	无	生成适用于 R8C/Tiny 系列的代码。
-R8CE	无	生成适用于 R8C/Tiny （ROM 64K 版本）系列的代码。

-fansi

修改所生成的代码

功能：　验证以下命令行选项：

-fnot_reserve_asm：	从保留字中删除 asm
-fnot_reserve_far_and_near：	从保留字中删除 far 和 near
-fnot_reserve_inline：	从保留字中删除 inline
-fextend_to_int：	将 char 类型数据转换为 int 类型，以进行操作

补充：　指定此选项后，编译器将遵循 ANSI 标准生成代码。

-fchar_enumerator　-fCE

修改所生成的代码

功能：　将 enumerator 类型处理为 unsigned char 类型，而不是 int 类型。

注意：　类型调试信息不包含类型大小的信息。
因此，如果指定了此选项， enum 类型可能在某些调试器中不能被正确引用。

-fconst_not_ROM -fCNR

修改所生成的代码

功能： 不会将 const 指定的类型作为 ROM 数据处理。

补充： 以 const 指定的数据根据默认设置定位在 ROM 区中。请参阅以下范例。

```
int   const   array[10] = { 1,2,3,4,5,6,7,8,9,10 };
```

在此情况下，数组 “array”被定位在 ROM 区中。通过指定此选项，您可以将 “array”定位在 RAM 区中。
您一般不需要使用此选项。

-fdouble_32 -fD32

修改所生成的代码

功能： 此选项指定将 double 类型像 float 类型一样，以 32 位的数据长度处理。

补充：
1. 若要使用此选项，必须明确的编写函数的原型。若没有原型声明，编译器可能无法生成正确的代码。
2. 指定此选项后， double 类型的调试信息将当作 float 类型处理。因此， double 类型的数据会在 C 监视窗口及 Debug （调试）工具的全局窗口中显示为 float 类型。

-fenable_register -fER

寄存器存储类

功能： 将带有指定寄存器存储类的变量分配至寄存器。

补充： 当优化 auto 变量的寄存器分配时，并非总能取得最佳的程序执行效果。此选项通过指示程序在上述情况下的寄存器分配，来提高优化的效率。
指定此选项后，以下指定了寄存器的变量将被强行分配到寄存器：
- 整数类型变量
- 指针变量

注意： 由于寄存器的指定在某些情况下会产生负面效果从而使效率降低，因此在使用此指定前，请确保先验证所生成的汇编语言。

-fextend_to_int -fETI

修改所生成的代码

功能： 将 char 类型和 signed char 类型的数据扩展为 int 类型的数据，以便执行操作 （根据 ANSI 的规则进行扩展）。

补充： 在 ANSI 标准中， char 类型和 signed char 类型的数据在运算时将总是扩展为 int 类型的数据。这项扩展是为了防止 char 类型的算术运算出现问题，例如在 c1 = c2 * 2 / c3 中， char 类型在运算中溢出，并导致了预料之外的值。如下例所示：

```
void    main(void)
{
        char    c1:
        char    c2 = 200;
        char    c3 = 2;

        c1 = c2 * 2 / c3;
}
```

在这种情况下，计算 [c2 * 2] 时 char 类型数据将溢出，因此可能返回错误的结果。
指定此选项有利于返回正确的值。默认情况下禁止扩展为 int 类型的原因是，这样做将有利于进一步提升 ROM 的效率。

-ffar_RAM -fFRAM

修改所生成的代码

功能： 将 RAM 数据的默认属性更改为 far。

补充： 默认情况下， RAM 数据（变量）定位在 near 区中。如果您希望将 RAM 数据定位在 near 区（64K 字节区）以外的其它区中，请使用此选项。

-finfo

修改所生成的代码

功能： 输出 “TM”、“STK Viewer”、“Map Viewer”和 “utl30”所需的信息。

补充： 在使用 “STK Viewer”、“Map Viewer”或 “utl30”时，将需要由此选项输出的绝对模块文件 “.x30”。

注意： 系统将不检查 asm 函数中全局变量的使用情况。因此，即使是在 “utl30”中， asm 函数的使用情况也将被忽略。
-finfo includes -g.

-fJSRW

修改所生成的代码

功能： 将调用函数的默认指令更改为 JSR.W。

补充： 如果调用的函数已定义为源文件的外部函数，则默认情况下将使用 “JSR.A” 命令。此选项允许将它更改为 “JSR.W” 命令。更改为 “JSR.W” 命令有助于压缩生成的代码大小。
如果程序相对较小，未超过 32K 字节，或需要压缩 ROM 时，可使用此选项。

注意： 相反地，如果调用的函数位于调用位置的前或后 32K 字节，则在连接时，“JSR.W” 命令将导致错误。与 “#pragma JSRA” 结合使用则可避免此错误。

-fbit -fB

修改所生成的代码

功能： 生成假设可以通过对映射到 near 区的所有外部变量使用绝对寻址，来执行按位操作指令的代码。

补充： 若将进行位操作的 near 区外部变量定位在 0000h 到 1FFFh 的 M16C 存储空间中，此选项的指定可帮助提高编译器所生成的代码效率。
在单芯片应用中，若 RAM 定位在上述存储空间内，将能证实此选项的指定有效。若尝试操作定位在任何其它存储空间内的变量，则将在连接时发生错误。

-fno_carry -fNC

修改所生成的代码

功能： 当使用 far 类型指针间接存取数据时，将禁止进位标志的加法。

补充： 当使用 far 类型指针间接存取结构或 32 位数据时，此选项将假设数据映射未跨越 64K 字节的边界，而生成不会对 far 类型指针 （32 位指针）的高 16 位执行进位加法的代码。因此，代码将更有效率。

注意： 当 far 类型的指针通过 malloc 函数等动态分配间接存取存储器或映射至 far 区的 ROM 数据时，请确保对数据的存取未跨越 64K 字节的边界。
此选项不能与 “-R8C” 选项同时使用。

-fauto_128 -fA1

修改所生成的代码

功能： 将可用堆栈帧限制为 128 字节。堆栈帧的最大大小默认值为 255 字节。

-ffar_pointer -fFP

更改所生成的代码

功能： 将 pointer 类型变量的默认属性更改为 far。
此选项将默认的指针大小设置为 32 位。

补充：
1. 本编译器的指针类型变量具有 near 属性的默认属性。此选项在将指针类型变量的默认属性更改为 far 属性时使用。
2. 描述 near 修饰符的指针变量不受此选项影响。它将总是变成 near 属性。
   范例：

```
char near *p; // 它作为 near 指针来处理。
```

-fnear_ROM -fNROM

修改所生成的代码

功能： 将 ROM 数据的默认属性更改为 near。

补充： 默认情况下， ROM 数据 （const 指定的变量等）被定位在 far 区域中。通过指定此选项，您可以将 ROM 数据定位在 near 区域中。
您一般不需要使用此选项。

-fno_align -fNA

修改所生成的代码

功能： 不对齐函数的开始地址。

-fno_even　　-fNE

修改所生成的代码

功能： 输出数据时，不分开奇数和偶数数据。也就是说，所有数据将被映射至奇数段 （data_NO、data_FO、 data_INO、 data_IFO、 bss_NO、 bss_FO、 rom_NO、 rom_FO）。

补充： 默认情况下，奇数大小和偶数大小数据分别被输出至单独的段。
请参阅以下范例。

```
char    c;
int     i;
```

在这种情况下，变量 “c”和变量 “i”将被输出到单独的段。这是因为偶数大小的变量 “i”位于偶数地址。如此一来，以 16 位总线宽度存取的速度就会更快。
仅在以 8 位总线宽度使用编译器，并且想要减少段数的情况下使用此选项。

注意： 当使用 “#pragma SECTION”来更改段名时，数据将被映射到新命名的段。

-fno_switch_table　　-fNST

修改所生成的代码

功能： 指定此选项时，在比较之后发生转移的代码将生成到一个 switch 语句中。

补充： 只有当未指定此选项而代码大小变小时，才会生成使用跳转表的代码。

-fnot_address_volatile　　-fNAV

修改所生成的代码

功能： 不将由 “#pragma ADDRESS”或 “#pragma EQU”指定的全局变量，或在函数外声明的 static 变量，作为由 volatile 指定的变量处理。

补充： 如果 I/O 变量进行了与 RAM 中变量相同的优化，则编译器可能不会按预期运行。只要为 I/O 变量指定 volatile 便可以避免此问题。
一般情况下， #pragma ADDRESS 或 #pragma EQU 会在 I/O 变量上运行，因此即使未指定 volatile，编译器也会假设已指定 volatile 来进行处理。此选项将禁止这样的处理。

注意： 您一般不需要使用此选项。

-fnot_reserve_asm -fNRA

修改所生成的代码

功能： 从保留字列表中删除 asm。

补充： 具备相同功能的 “_asm”将作为保留字处理。

-fnot_reserve_far_and_near -fNRFAN

修改所生成的代码

功能： 从保留字列表中删除 far 和 near。

补充： 具备相同功能的 “_far”和 “_near”将作为保留字处理。

-fnot_reserve_inline -fNRI

修改所生成的代码

功能： 不将 inline 当作保留字处理。

补充： 具备相同功能的 “_inline”将作为保留字处理。

-fsmall_array -fSA

修改所生成的代码

功能： 引用 far 类型的数组时，如果在编译时不知其总大小，此选项将假设数组总大小低于 64K 字节，并以 16 位计算数组下标。

补充： 在引用 far 类型数组中的数组元素时，例如 ROM 中的数组数据，若不知 far 类型数组的总大小，编译器将以 32 位计算数组下标，以便能够处理大小在 64K 字节或以上的数组。
请参阅以下范例。

```
extern  int   array[]:
int           i = array[j];
```

在这种情况下，由于编译器不知数组 array 的总大小，因此将以 32 位计算下标 “j”。
在指定此选项后，编译器将假设数组总大小等于或低于 64K 字节，并以 16 位计算下标 “j”。结果，处理速度得以提高，代码大小也相应减小。
瑞萨建议在一个数组的大小不超过 64K 字节时使用此选项。

-fswitch_other_section -fSOS

修改所生成的代码

功能： 此选项将 ‘switch’ 语句的 ROM 表输出至其它段，而不是程序段。

补充： 段名为 ‘switch_table’

注意： 一般不需要使用此选项。

-fchange_bank_always -fCBA

修改所生成的代码

功能： 此选项允许您将多个变量写入扩展区 （使用 #pragma EXT4MPTR）。

补充： 当您将多个指针变量声明至 4M 字节空间，同时使用 #pragma EXT4MPTR 功能时，请指定此选项。

注意： 此选项不能与 “-R8C” 选项同时使用。

-fauto_over_255 -fAO2

修改所生成的代码

功能： 将每个函数可保留的堆栈帧大小更改为 64K 字节。
（堆栈帧默认的最大值为 255 字节。）

注意：

1. 此选项不能与 #pragma SBDATA 同时使用。若所编译的一个文件中具有 #pragma SBDATA 的描述，则将输出下面的警告，以及被忽略的 #pragma SBDATA 的描述。

   [Warning(ccom):XX.c,line XX] compile option -fauto_over_255 is specified, #pragma SBDATA was ignored. （已指定编译选项 - fauto_over_255，因此 #pragma SBDATA 被忽略。）

   ==> #pragma SBDATA xxx;

   由于 SB 寄存器被用于堆栈帧，因此不可使用 #pragma SBDATA。

2. 为下面描述的文件指定此选项。
   a. 当存在一个需要 255 字节或以上的堆栈帧的函数时 （在下文中称为函数 A）
   ==> 编写函数 A 的文件

   b. 当在处理函数 A 时发生中断 （在下文中称为中断 A），并从中断 A 存取一个通过 #pragma SBDATA 来声明的变量时
   ==> 编写中断 A 的文件

-fsizet_16 -fS16

更改类型定义的位大小

功能： 将 size_t 的类型定义从 unsigned long 类型更改为 unsigned int 类型。

注意： 若选择了此选项，请确保在连接时使用下面所列的其中一个标准函数程序库。

- M16C/60 系列
  nc30s16.lib
- R8C/Tiny 系列
  r8cs16.lib

-fptrdifft_16 -fP16

更改类型定义的位大小

功能： 将 ptrdiff_t 的类型定义从 signed long 类型更改为 signed int 类型。

注意： 若选择了此选项，请确保在连接时使用下面所列的其中一个标准函数程序库。

- M16C/60 系列
  nc30s16.lib
- R8C/Tiny 系列
  r8cs16.lib

-fuse_DIV -fUD

修改所生成的代码

功能： 此选项将更改除法运算的生成代码。

补充： 在除法运算中，如果被除数为 4 字节值，除数为 2 字节值且结果为 2 字节值，或者如果被除数为 2 字节值，除数为 1 字节值且结果为 1 字节值，则编译器将生成 div.w （divu.w）和 div.b （divu.b）单片机指令。

注意：
1. 指定此选项时，如果除法运算导致溢出，则编译器的运行可能与 ANSI 中的规定有所不同。
2. M16C 的 div 指令具备以下特点：如果运算导致溢出，则无法确定结果。因此，在 NC30 的默认设置中，当程序被编译时，运行时程序库将被调用，以更正此问题的结果 （即使在被除数为 4 字节，除数为 2 字节且结果为 2 字节的情况下）。

-fuse_MUL -fUM

修改所生成的代码

功能： 此选项将更改乘法运算的生成代码。

补充： 当在 32 位中存储 16 位 x16 位时，由于它获取高 16 位的结果，它应在 32 位的乘数或被乘数中转换类型 （Cast）。
32 位的结果可通过指定 Cast 选项来获取。

-R8C

修改所生成的代码

功能： 生成适用于 R8C/Tiny 系列的代码。

补充： _fnear_ROM （-fNROM）被默认设置。

注意： 此选项不能与下列选项同时使用。
若指定了下列选项之一，该选项将被忽略。
-ffar_RAM(- fFRAM)、 -fno_carry(- fNC)、 -fchange_bank_always(- fCBA)

-R8CE

修改所生成的代码

功能： 生成适用于 R8C/Tiny （ROM 64K 版本）系列的代码。

注意：
1. 此选项不能与下列选项同时使用。若指定了下列此选项，该选项将被忽略。
   -fchange_bank_always(- fCBA)
2. 当 ROM 区超出 64K 的边界时，使用此选项。

-fSB_auto -fSBA

修改所生成的代码

功能： 从一个接一个函数中切换 SB 寄存器，生成 SB 相对寻址。

补充： 分析外部变量在函数中被引用的次数，并从一个接一个函数中生成最佳的 SB 相对寻址。

```
int sym;
int a;
int data;
   :
int b;
   :
int func(void){
        a = x;                                  _sym 的地址将作为 SB 相对寻址的基点。
        sym = xx;
        sym = a * b;
        if(sym != 0)
                sym = sub();
        return sym;
}
int data1,data2;
int sub(void)                                   _data1 的地址将作为 SB 相对寻址的基点。
{
        data1 = sym1;
        data2 = data1/2;
        data1 = sub1(data2);
                :
                :
}
```

1. 作为 SB 相对寻址的基点的符号地址将存储在 SB 寄存器中。
2. 保存 / 恢复 SB 寄存器的代码，将在进入和退出函数时生成。
3. 只有外部变量有效。
4. 此选项不能与 -OR、 -OS、 -OR_MAX 及 -OS_MAX 一同使用。

附录 A.2.7 程序库的指定选项

附表 A.9 列出可用来指定一个程序库文件的启动选项。

附表 A.9 程序库的指定选项

选项	功能
-l *程序库文件名*	指定 ln30 在连接文件时使用的程序库文件。

-l *程序库文件名*

指定一个程序库文件

功能： 指定 ln30 在连接文件时使用的程序库文件。文件扩展名可以省略。

语法： nc30Δ-l *文件名*Δ<*C 源文件名称*>

注意：
1. 在文件指定中，可以省略扩展名。如果省略了文件扩展名，则在对文件进行处理时将假设其扩展名为 “.lib”。
2. 若您指定文件扩展名，请确保指定 “.lib”。
3. 默认情况下， NC30 将连接在环境变量 LIB30 中指定的目录中的程序库 “nc30lib.lib”。（如果您指定了多个程序库， nc30lib.lib 则将在引用时被指定为最低的优先级。）
4. 若指定了多个程序库，对 “nc30lib.lib”的引用将被指定为最低的优先级。

附录 A.2.8 警告选项

附表 A.10 显示用于在违反 nc30 语言的指定时输出警告信息的命令行选项。

附表 A.10 警告选项

选项	缩写	功能
-Wall	无	为所有可检测的警告显示信息。 （不包括 -Wlarge_to_small 和 “-Wno_used_argument”所输出的警报）
-Wccom_max_warnings = *警告计数*	-WCMW	此选项允许您指定由 ccom30 输出的警告信息数的上限。
-Werror_file< *文件名* >	-WEF	将错误信息输出至指定的文件。
-Wlarge_to_small	-WLTS	按大小次序输出有关变量隐性转换的警告信息。
-Wmake_tagfile	-WMT	按照每一个源文件的标志文件输出错误和警告信息。
-Wnesting_comment	-WNC	对含有 “*/”的注释输出警告。
-Wno_stop	-WNS	防止出现错误时停止编译器。
-Wno_used_argument	-WNUA	为未使用的函数参数输出警告信息。
-Wno_used_function	-WNUF	在连接时显示未使用的全局函数。
-Wno_used_static_function	-WNUSF	显示不需要生成代码的静态函数名称。
-Wno_warning_stdlib	-WNWS	当已指定 “-Wnon_prototype”或 “-Wall”时，如果再指定此选项将会禁止 “Alarm for standard libraries which do not have prototype declaration”（没有原型声明的标准程序库的警告）。
-Wnon_prototype	-WNP	为没有原型声明的函数输出警告信息。
-Wstdout	无	将错误信息输出至主机的标准输出 （stdout）。
-Wstop_at_link	-WSAL	在连接时若发生警告，则停止连接源文件，以禁止绝对模块文件的生成。同时，返回值 “10”将返回到主机 OS。
-Wstop_at_warning	-WSAW	在编译时若发生警告，则停止编译源文件，并返回结束代码 “10”。
-Wundefined_macro	-WUM	对在 #if 中使用未定义的宏发出警告。
-Wuninitialize_variable	-WUV	对尚未初始化的 auto 变量输出警告。
-Wunknown_pragma	-WUP	对不支持的 #pragma 输出警告信息。

-Wall

警告选项

功能： 显示所有可检测的警报。

补充：
1. 在此处显示的警报，不包括使用 “Wlarge_to_small(-WLTS)”和 “Wno_used_argument (-WNUA)”以及 “Wno_used_static_function(-WNUSF)”时所可能生成的警报。
2. 在此处显示的警报等同于 “Wnon_prototype(-WNP)”、“Wunknown_pragma(-WUP)”、“Wnesting_comment(-WNC)”和 “Wuninitialize_variable(-WUV)”的选项。
3. 警报也将在下列情况中显示：
   - 在 if 语句、 for 语句或比较语句 （带有 && 或 || 运算符）中使用了赋值运算符 “=”时。
   - 应指定 ‘=’但却写入了 “==”时。
   - 以旧格式定义函数时。

注意： 这些警报被检测的范围，是由编译器以错误的程序描述为假设判断的。因此，并非所有错误都会发出警报。

-Wccom_max_warnings= *警告计数* -WCMW= *警告计数*

警告选项

功能： 此选项允许您指定由 ccom30 输出的警告信息数的上限。

补充： 默认情况下，不规定警告信息输出数的上限。
当输出了大量警告信息而需要滚动屏幕时，可以使用此选项来进行调整。

注意： 对于警告信息输出数的上限计数，请指定一个等于或大于 0 的数字。此计数的指定不可省略。如果您指定了 0，则警告信息输出将会被完全禁止。

-Werror_file < *文件名* >

警告选项

功能： 将错误信息输出至指定的文件。

语法： nc30Δ-W*error_file*Δ< *输出错误信息的文件名* >

注意： 将错误信息输出至文件时所采用的格式与错误信息在屏幕中显示的格式是不同的。将错误信息输出至文件时，它们将以适合于某些编辑器所具有的 “tag jump function”（标签跳转功能）的格式输出。

-Wlarge_to_small -WLTS

警告选项

功能： 按大小次序输出有关变量替换的警告信息。

补充： 当出现任何类型的负数边界值时，都可能输出警告信息，尽管这些值是符合类型的。这是因为，依照语言惯例，负数值被看作是附加了一元运算符 （-）的整数。
例如，值 -32768 符合 signed int 类型，但是在拆分为 “-”和 “32768”时，值 32768 就不符合 signed int 类型，而成为 signed long 类型。
因此，立即值 32768 是 signed long 类型。基于这个原因，任何类似于 “int i = 32768 ； ”的语句都将引发警告信息。

注意： 由于此选项输出了大量警告信息，因此，下列类型转换的警告信息将被禁止输出。
- 从 char 类型变量到 char 类型变量的赋值
- 立即值到 char 类型变量的赋值
- 立即值到 float 类型变量的赋值

-Wmake_tagfile -WMT

警告选项

功能： 当出现错误或警告信息时，按照每一个源文件的标志文件输出错误和警告信息。

补充： 此选项不能与 “-Werror_file (-WEF)”选项一同指定。

-Wnesting_comment -WNC

警告选项

功能： 在注释包含 “/*”时，生成警告信息。

补充： 通过使用此选项，将可能检测注释的嵌套。

-Wno_stop -WNS

警告选项

功能： 防止出现错误时停止编译器。

补充： 对程序进行编译时，编译器一次只编译一个函数。如果在编译时出现错误，在默认情况下，编译器将不会继续编译下一个函数。
同时，一个错误可能引发另一个错误，最终导致多个错误的发生。在这种情况下，编译器将停止编译。
如果指定了此选项，编译器将尽可能编译更多函数。

注意： 由于程序中的错误描述，有可能出现系统错误。在这种情况下，即使指定此选项，编译器也将停止编译。

-Wno_used_argument -WNUA

警告选项

功能： 为未使用的函数参数输出警告信息。

-Wno_used_function -WNUF

警告选项

功能： 在连接时显示未使用的全局函数。

注意： 当选择此选项时，确保同时指定 “-finfo”选项。
当在连接过程中指定了 -U 选项时，则不需要此选项。

-Wno_used_static_function -WNUSF

警告选项

功能： 由于以下原因之一，输出了不需要生成代码的静态函数名称。

- 通过使用 “-Ostatic_to_inline(-OSTI)”选项，静态函数将被直接插入。
- 未从文件的任何一处引用静态函数。

注意： 1. 若有任何函数名称编写在以下面所示的方式进行初始化的数组中，编译器将在假设该函数将被引用的情况下处理函数，即使它在程序执行过程中并不会真的被引用。

范例：

```
void    (*a[5])(void) = {f1,f2,f3,f4,f5};

        for(i = 0; i < 3; i++) (*a[i])();
```

* 在上面的范例中，虽然函数 f4 和 f5 未被引用，但编译器将在假设这些函数会被引用的情况下来处理它们。

-Wno_warning_stdlib -WNWS

警告选项

功能： 当已指定 “-Wnon_prototype”或 “-Wall”时，如果再指定此选项将会禁止 “Alarm for standard libraries which do not have prototype declaration”（没有原型声明的标准程序库的警告）。

-Wnon_prototype -WNP

警告选项

功能： 为没有原型声明的函数，或在未向任何函数作出原型声明时，输出警告信息。

补充： 通过编写原型声明，函数参数可通过寄存器传递。
通过寄存器传递参数可提升速度，并减小代码大小。另外，原型声明也将使编译器检查函数参数，从而增强程序的可靠性。
所以，瑞萨建议您尽可能使用此选项。

-Wstdout

警告选项

功能： 将错误信息输出至主机的标准输出 （stdout）。

补充： 通过此选项，可将错误输出等使用 Redirect 保存到文件。

注意： 在本编译器中，由编译驱动器所引起的汇编器与连接编辑器错误，无论是否指定此选项，都会输出到标准输出。

-Wstop_at_link -WSAL

警告选项

功能： 在连接时若发生警告，则停止连接源文件，以禁止生成绝对模块文件。同时，返回值 “10”将返回到主机 OS。

-Wstop_at_warning -WSAW

警告选项

功能： 在编译时若发生警告，则停止编译源文件，并返回编译器结束代码 “10”。

补充： 如果编译时出现了警告信息，默认情况下，编译将被终止，结束代码为 “0”（正常终止）。
如果您正在使用 make 工具等，并希望在出现警告信息时停止编译处理，则请使用此选项。

-Wundefined_macro -WUM

警告选项

功能： 对在 #if 中使用未定义的宏发出警告。

-Wuninitialize_variable -WUV

警告选项

功能： 为没有初始化的 auto 变量输出警告信息。
即使指定 “-Wall”，此选项仍然有效。

补充： 如果 auto 变量被用户应用中的 if 或 for 等语句以条件跳转初始化，编译器将假设其尚未初始化。
因此，使用此选项时，编译器将为其输出警告信息。

-Wunknown_pragma -WUP

警告选项

功能： 对不支持的 #pragma 输出警告信息。

补充： 默认情况下，即便使用了不支持且未知的 “#pragma”，系统也不会发出警报。
如果您只使用 NC 系列的编译器，使用此选项将有助于查找 “#pragma”中的拼写错误。

注意： 如果您只使用 NC 系列的编译器，瑞萨建议您在编译时总是使用此选项。

附录 A.2.9 汇编和连接选项

附表 A.11 显示指定 as30 和 ln30 的选项的命令行选项。

附表 A.11 汇编和连接选项

选项	功能
-as30Δ< 选项 >	为 as30 连接命令指定选项。如果指定两个或多个选项，则用双引号将它们括起来。
-ln30Δ< 选项 >	为 ln30 汇编命令指定选项。如果指定两个或多个选项，则用双引号将它们括起来。

-as30 “*选项*”

汇编 / 连接选项

功能： 指定 as30 汇编命令选项。
如果指定两个或多个选项，则用双引号将它们括起来。

语法： nc30Δ-as30Δ “*选项 1Δ 选项 2*” Δ<*C 源文件*>

注意： 请勿指定 as30 选项 “-.”、“-C”、“-M”、“-O”、“-P”、“-T”、“-V” 或 “-X”。

-ln30 “*选项*”

汇编 / 连接选项

功能： 为 ln30 连接命令指定选项。您可以指定最多四个选项。
如果指定两个或多个选项，则用双引号将它们括起来。

语法： nc30Δ-ln30Δ “*选项 1Δ 选项 2*” Δ<*C 源文件名称*>

注意： 请勿指定 ln30 选项 “-.”、“-G”、“-O”、“-ORDER”、“-L”、“-T”、“-V” 或 “@ file”。

附录 A.3 有关命令行选项的注意事项

附录 A.3.1 编码命令行选项

NC30 命令行选项的编写有大小写区分。
若以错误的大小写来编写，可能导致有些选项无效。

附录 A.3.2 选项的控制优先级

若您在 NC30 命令行中指定了下列两个选项， -S 选项将具有优先权，同时将只生成汇编语言的源文件。

- “-c”：在建立可再定位的模块文件后停止。
- “-S”：在建立汇编语言的源文件后停止。

附录 B　扩展功能参考

为方便 M16C/60、 M16C/30、 M16C/Tiny、 M16C/20、 M16C/10 R8C/Tiny 系列在系统中的使用， NC30 提供了一系列的附加 （扩展）功能。

本附录 B 将描述这些扩展功能的使用，但对于和语言指定相关的部分，则将只在概述中说明。

附表 B.1　　扩展功能 (1)

扩展功能	描述
near/far 修饰符	指定存取数据的寻址模式。 near..... 存取 64K 字节内的区 （0H 至 0FFFFH）。 far..... 存取 64K 字节以外的区 （所有存储区）。 * 所有函数均具有 far 属性。

asm 函数	1. 可将汇编语言直接包含在 C 程序中。它也可以被包含在函数外。
	范例：asm(" MOV.W #0, R0");
	2. 您可以指定变量名称 （仅限于函数内）。
	范例 1：
	asm(" MOV.W R0, $$[FB]",f);
	范例 2：
	asm(" MOV.W R0, $$",s);
	范例 3：
	asm(" MOV.W R0, $@",f);
	3. 您可以包含虚设的 asm 函数，以部分地禁止优化 （仅限于函数内）。
	范例：asm( );

日文字符	1. 允许您在字符串中使用日文字符。 范例 1： L “ 漢字 ” 2. 允许您对字符常数使用日文字符。 范例： L “ 漢 ” 3. 允许您在注释中书写日文字符。 范例： /* 漢字 */ * 支持 Shift-JIS 和 EUC 代码，但无法使用日文片假名 (Katagana) 的半角字符。

函数的默认参数声明	可为函数的参数定义默认值。
	范例 1：
	extern int func(int=1, char=0);
	范例 2：
	extern int func(int=a, char=0);
	* 当编写一个变量作为默认值时，确保在声明函数前，先声明该用作默认值的变量。
	* 紧接着参数之后开始按顺序编写默认值。

直接插入存储类	可通过直接插入存储类来指定直接插入函数 specifier.inline. 范例： inline func(int i); * 总是确保在使用直接插入函数前，先定义该直接插入函数的主体。

附表 B.2　　扩展功能 (2)

扩展功能	描述
注释的扩展	您可以加插类似 C++ 的注释 （“//”） 范例： // 这是一项注释。
#pragma 扩展功能	您可以为 M16C/60、M16C/30、M16C/Tiny、M16C/20、M16C/10 R8C/Tiny 系列的硬件使用 C 语言的扩展功能。

宏汇编器函数	您可以将一些汇编器命令描述为 C 函数
	范例：
	char dadd_b(char val1, char val2);
	范例：
	int dadd_w(char val1, char val2);

附录 B.1 near 和 far 修饰符

对于 M16C/60 系列的单片机，用于引用和定位数据的寻址模式在边界地址 0FFFFH 上有所不同。NC30 允许您通过切换 near 和 far 修饰符来控制寻址模式。

附录 B.1.1 near 和 far 修饰符的概述

near 和 far 修饰符选择一个用于变量或函数的寻址模式。

* near 修饰符 ..... 000000H 到 00FFFFH 的区
* far 修饰符 ..... 000000H 到 0FFFFFH 的区

在声明变量或函数时，一个类型说明符将添加 near 和 far 修饰符。若您在声明变量和函数时未指定 near 或 far 修饰符， NC30 将按下列规则来理解其属性：

* 变量 ..... near 属性
* 限定为 const 的常数 ..... far 属性
* 函数 ..... far 属性

另外， NC30 允许您通过使用编译驱动器 nc30 的启动选项来修改这些默认属性。

附录 B.1.2 变量声明的格式

near 及 far 修饰符使用和 const 及 volatile 类型修饰符相同的句法格式，被包含在声明中。附图 B.1 是变量声明的格式。

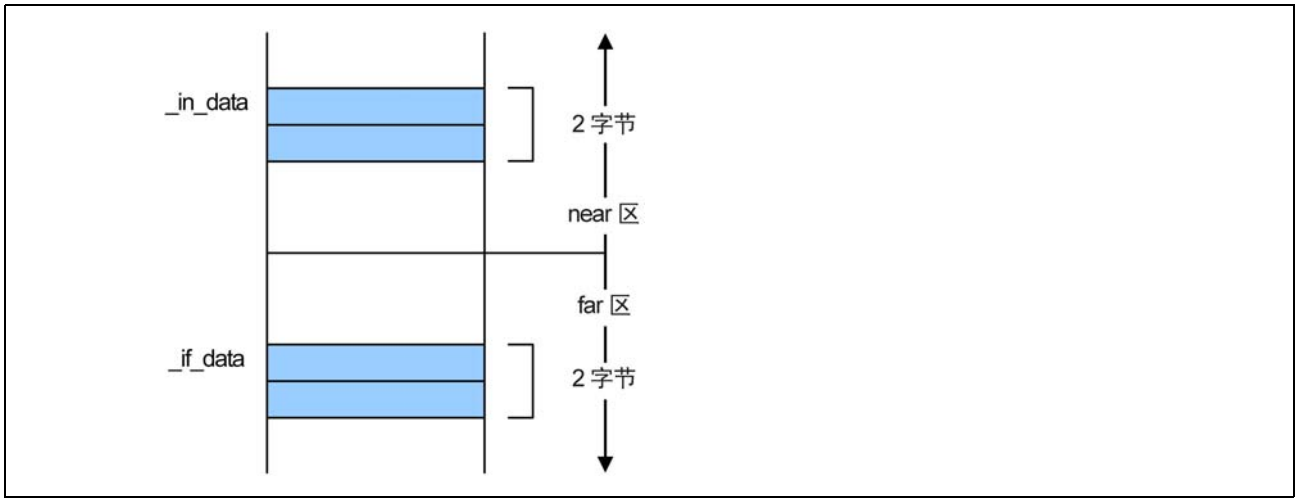
```
类型说明符△ near 或 far △变量；
```

附图 B.1 添加了 near/far 修饰符的变量格式

附图 B.2 是变量声明的一个范例。附图 B.3 是该变量的存储器映像。

```
int near  in_data;
int far   if_data;

 void     func(void)
{
          （其余省略）
            :
```

附图 B.2 变量声明的范例



附图 B.3　　变量的存储器位置

附录 B.1.3　　指针类型变量的格式

指针类型变量默认情况下为 near 类型 （2 字节）变量。 附图 B.4 中显示一个指针类型变量的声明范例。

```
范例：
        int     * ptr;
```

附图 B.4　　声明指针类型变量的范例 (1)

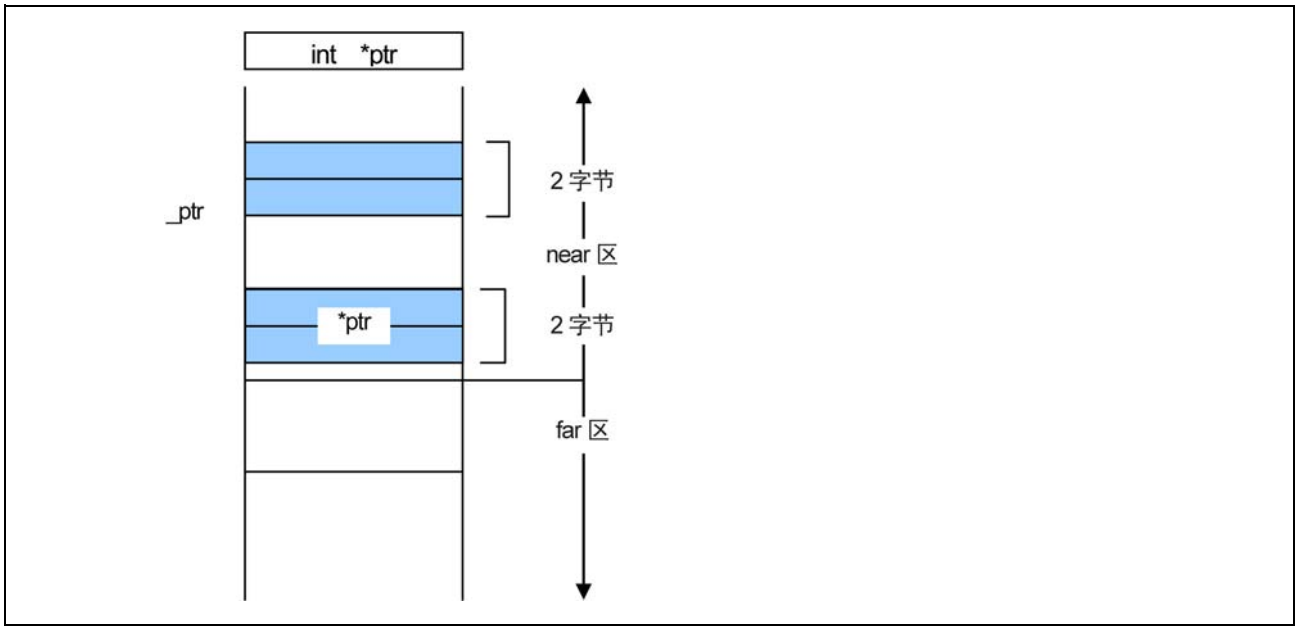
由于变量定位在 near，同时具有 near 的指针变量类型，因此对附图 B.4 中描述的理解如附图 B.5 所示。

```
范例：
        int     near*  near   ptr;
```

附图 B.5　　声明指针类型变量的范例 (2)

变量 ptr 是一个 2 字节的变量，表示 int 类型变量定位在 near 区中。ptr 本身定位在 near 区中。
上述范例的存储器映像如附图 B.6 中所示。

附图 B.6 显示上述范例的存储器映射。



附图 B.6　　指针类型变量的存储器位置

当明确地指定了 “near and far”时，必须确定用来存储右侧编写的 “variable and function”的地址大小。附图 B.7 中显示处理地址的指针类型变量的声明。

```
范例 1：
        int     far *       ptr1;
范例 2：
        int     * far       ptr2;
```

附图 B.7　　声明指针类型变量的范例 (1)

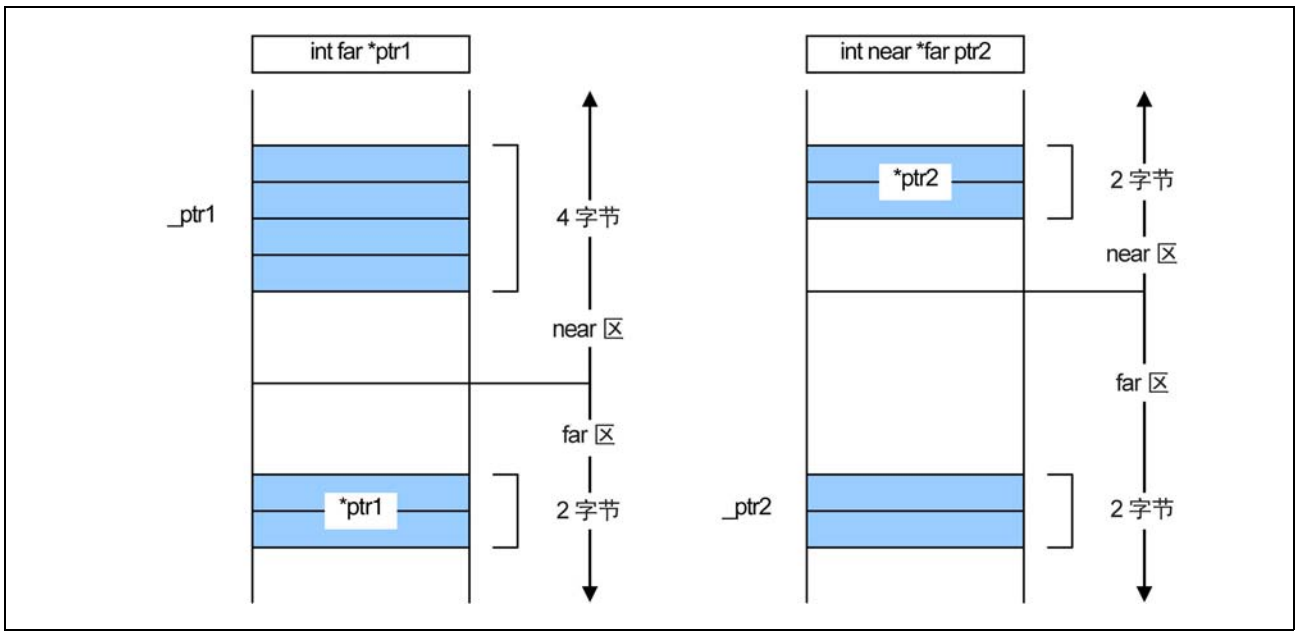
如之前所述，除非指定了 “near and far”，否则编译器将把变量位置视为 “near”，同时把变量类型视为 “far”来处理。因此，范例 1 和 2 分别具有如附图 B.8 所示的理解。

```
范例 1：
        int     far *near   ptr1;
范例 2：
        int     near * far  ptr2;
```

附图 B.8　　声明指针类型变量的范例 (2)

在范例 1 中，变量 ptr1 是一个 4 字节的变量，表示 int 类型变量定位在 far 区中。变量本身定位在 near 区中。在范例 2 中，变量 ptr2 是一个 2 字节的变量，表示 int 类型变量定位在 near 区中。变量本身定位在 far 区中。

附图 B.9 中显示范例 1 和 2 的存储器映像。



附图 B.9　　指针类型变量的存储器位置

附录 B.1.4　　函数声明的格式

函数的 near 和 far 分配属性总是为 far。若您在函数声明中指定了 near 属性，系统将输出警告信息（function must be far （函数必须为 far）），并忽略您的 near 声明。

附录 B.1.5　　通过 nc30 命令行选项控制 near 和 far

若您未指定 near 和 far 的属性，NC30 将把函数的属性作为 far 及将变量作为 near 处理。NC30 的命令行选项允许您修改函数及变量 （数据）的默认属性。这些内容在下表中列出。

附表 B.3　　命令行选项

命令行选项	功能
-fnear_ROM(-fNROM)	假设 near 是 ROM 数据的默认属性。
-ffar_RAM(-fFRAM)	假设 far 是 RAM 数据的默认属性。

附录 B.1.6 从 near 到 far 类型转换的功能

附图 B.10 中的程序执行了从 near 到 far 的类型转换。

```
int     func( int far * );
int     far *f_ptr;
int     near *n_ptr;

void    main(void)
{
        f_ptr = n_ptr;                  /* 将 far 指针赋值为 near 指针的值 */
            :
        （已省略）
            :
        func ( n_ptr );                 /* 将函数的原型声明的参数赋值为 far 指针 */
                                        /* 对函数调用指定 near 指针参数 */
}
```

附图 B.10 从 near 到 far 的类型转换

当将类型转换为 far 时， 0 （零）将扩展为高顺序地址。

附录 B.1.7 检查将 far 指针赋值到 near 指针的函数

在编译时，会为附图 B.11 中的代码输出 “assign far pointer to near pointer, bank value ignored”（将 far 指针赋值到 near 指针，存储体的值已忽略）的警告信息，显示地址的高位 （存储体的值）已丢失。

```
int     func( int near * );
int     far *f_ptr;
int     near *n_ptr;

void    main(void)
{
        n_ptr = f_ptr;                  /* 将 near 指针赋值为 far 指针的值 */
            :
        （已省略）
            :
        func ( f_pyr );                 /* 函数的原型声明 */
                                        /* 通过参数中的 near 指针 */
                                        /* far 指针隐含地转型为 near 类型 */

        n_ptr = (near *)f_ptr;          /* far 指针明确地转型为 */
                                        /* near 类型 */
}
```

附图 B.11 从 far 到 near 的类型转换

当 far 指针被明确地转型为 near 指针，然后赋值到 near 指针时，输出 “far pointer (implicitly) casted by near pointer”（由 near 指针 （隐含地）转型为 far 指针）的警告信息。

附录 B.1.8 声明函数

在 NC30 中，函数总是定位在 far 区内。因此，请勿为函数编写 near 声明。

若一个函数被声明为具有 near 属性， NC30 将输出警告，并假设该函数的属性为 far 来继续处理。附图 B.12 显示一个函数被声明为 near 的范例显示。

```
%nc30  -S  smp.c
M16C/60 Series NC30 COMPILER V.X.XX Release XX
Copyright(C) XXXX(XXXX-XXXX). Renesas Technology Corp.
and Renesas Solutions Corp., All rights reserved.
smp.c
[Warning(ccom):smp.c,line 3] function must be far
===> {
func
%
```

附图 B.12 函数声明的范例

附录 B.1.9 在多个声明中指定 near 和 far 的函数

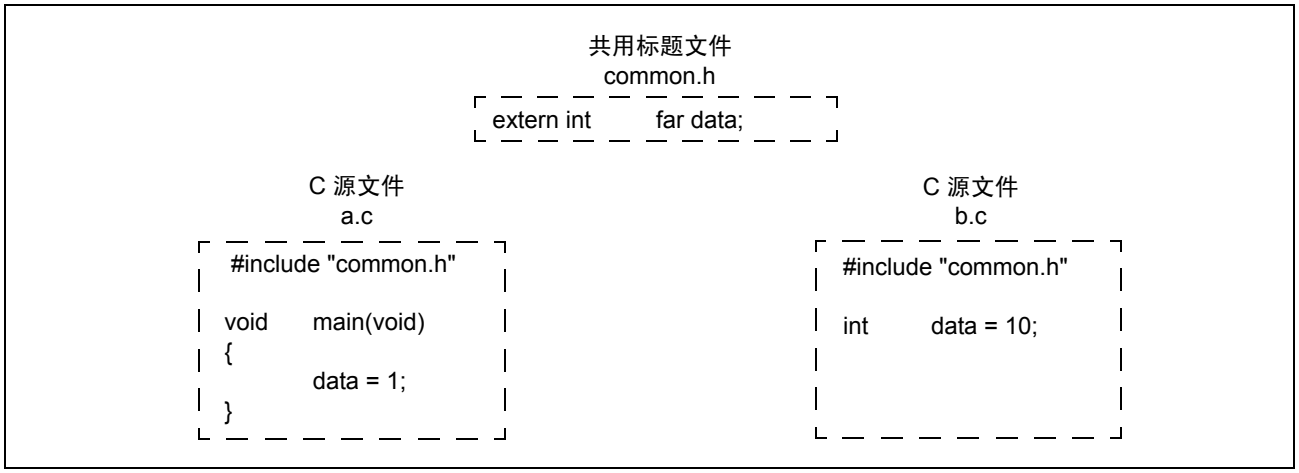
如附图 B.13 中所示，若相同的变量具有多个声明，该变量的类型信息将被理解为表示一个复合类型。

```
extern int      far idata;
int             idata;
int             idata = 10;

void            func(void)
{
                （其余省略）
                    :


对此声明的理解如下：

extern int      far idata = 10;

void            func(void)
{
                （其余省略）
                    :
```

附图 B.13 变量声明的综合功能

如本范例中所示，若有多个声明，则可在这些声明中的其中一个，指定 “near or far”，来声明类型。不过，若在两个或多个这些声明中的 near 和 far 指定之间存在任何冲突，则发生错误。

可以使用一个共用标题文件来声明 “near or far”，以确保源文件的一致性。



附图 B.14　　共用标题文件声明的范例

附录 B.1.10　　有关 near 和 far 属性的注意事项

(a)　有关函数的 near 和 far 属性的注意事项

函数总是假设具有 far 的属性。请勿为函数声明 near。若为函数声明 near 属性， NC30 将会输出警告。

(b)　有关 near 和 far 修饰符语法的注意事项

near 和 far 修饰符与 const 修饰符在语法上相同。因此下列代码会产生错误。

```
int     i, far      j;          ← 这是不允许的
    ▼
int     i;
int     far j;
```

附图 B.15　　变量声明的范例

附录 B.2 asm 函数

NC30 允许在 C 源程序中包含汇编语言例程 （asm 函数）[1]。

附录 B.2.1 asm 函数的概述

asm 函数被用来将汇编语言代码包含在 C 源程序中。如附图 B.16 中所示， asm 函数的格式是 asm(" ");，遵循 AS30 语言规范指定的汇编语言指令将包含在双引号中。

```
#pragma  ADDRESS  ta0_int  55H
char     ta0_int;

void     func(void)
{
           :
         （已省略）
           :
         ta0_int = 0x07;              ← 允许计时器 A0 中断

         asm("  FSET I");             ← 设置中断允许标志
}
```

附图 B.16 asm 函数的描述范例 (1/2)

根据语句的位置关系进行的编译器优化，可通过使用附图 B.17 中所示的代码来部分地禁止。

```
asm( );
```

附图 B.17 编写 asm 函数的范例 (2/2)

NC30 中所使用的 asm 函数不仅可让您包含汇编语言代码，同时还可拥有以下扩展功能：

- 使用变量的 C 名称来指定 C 程序存储类 auto 变量的 FB 偏移
- 使用变量的 C 名称来指定 C 程序存储类 register 变量的寄存器名称
- 使用变量的 C 名称来指定 C 程序存储类 extern 与 static 变量的符号名称

下面显示使用 asm 函数时所必须注意的事项。

- 请勿销毁 asm 函数中的寄存器内容。
  编译器并不会检查 asm 函数的内部。若寄存器的内容将会被销毁，则使用 asm 函数来编写 push 和 pop 指令，以保存及恢复寄存器。

[1] 为符合本用户手册中的表达方式，用汇编语言编写的子例程称为汇编函数。在 C 语言程序中用 asm() 编写的函数则称为 asm 函数或直接插入的汇编描述。

附录 B.2.2 指定 auto 变量的 FB 偏移值

以 C 语言编写的存储类的 auto 及 register 变量 （包括参数）将作为帧基址 （Frame Base 或简称 FB）寄存器的偏移被引用及定位。（它们可能因为优化的结果被映射到寄存器。）

通过按照下面附图 B.18 中所示的方法来编写程序，将可在 asm 函数中使用被映射到堆栈的 auto 变量。

```
asm( "          优化代码      R1 , $$ [ FB ] " , 变量名称 );
```

附图 B.18 指定 FB 偏移的描述格式

使用此描述格式，只能指定两个变量名称。下面是所支持的变量名称类型：

- 变量名称
- 数组名称 [整数]
- 结构名称，成员名称 （不包括位字段成员）

```
 void    func(void)
{
        int             idata;
        int             a[3];
        struct TAG{
                        int             i;
                        int             k;
        } s;
        :
        asm("           MOV.W       R0, $$[FB]", idata);
                            :
        asm("           MOV.W       R0, $$[FB]", a[2]);
                            :
        asm("           MOV.W       R0, $$[FB]", s.i);
                        （其余省略）
                            :
        asm("           MOV.W       $$[FB], $$[FB]", s.i, a[2]);
}
```

附图 B.19 指定的描述范例

附图 B.20 显示引用 auto 变量及其编译结果的范例。

- C 源文件：

```
void    func(void)
{
        int idata = 1;                          ← auto 变量 （FB 偏移值 =-2）

        asm("               MOV.W      $$[FB], R0", idata);
        asm("               CMP.W      #00001H ,R0");
        （其余省略）
              :
}
```

- 汇编语言源文件 （编译结果）：

```
;## # FUNCTION func
;## # FRAME AUTO ( idata) size 2, offset -2
                :
          （已省略）
;## # C_SRC : asm("          MOV.W      $$[FB], R0", idata);
;#### ASM START
        MOV.W               -2[FB], R0      ← 将 FB 偏移值 -2 转移到 R0 寄存器
        ._line 5
;## # C_SRC : asm(" CMP.W #00001H,R0");
        CMP.W               #00001H ,R0
;#### ASM END
        （其余省略）
           :
```

附图 B.20　　引用 auto 变量的范例

也可以使用附图 B.21 中的格式，这样一来，asm 函数中的 auto 变量就将使用 1 位的字段。（无法操作大于 2 位的位字段）。

```
asm( "   优化代码      $b[ FB ]" , 位字段名称 );
```

附图 B.21　　FB 偏移位的位置的指定格式

只能使用这个格式来指定一个变量名称。附图 B.22 是一个范例。

```
 void    func(void)
{
        struct TAG{
                char        bit0:1;
                char        bit1:1;
                char        bit2:1;
                char        bit3:1;
        } s;

        asm("       bset        $b[FB]",s.bit1);
}
```

附图 B.22　　指定 FB 偏移位置的范例

附图 B.23 显示引用 auto 区的位字段及编译结果的范例。

- C 源文件：

```
void    func(void)
{
        struct TAG{
                char            bit0:1;
                char            bit1:1;
                char            bit2:1;
                char            bit3:1;
        } s;
        asm("   bset            $b[FB]",s.bit1);
}
```

- 汇编语言源文件 （编译结果）：

```
;## # FUNCTION func
;## # FRAME AUTO ( __PAD1) size 1, offset -1
;## # FRAME AUTO ( s) size 1, offset -2
;## # ARG Size(0) Auto Size(2) Context Size(8)
        .section    program,CODE,ALIGN
        ._file      'bit.c'
        .align
        ._line 3
        .glb        _func
_func:
        enter       #02H
        ._line 10
;#### ASM START
        bset        1,-2[FB] ; s
;#### ASM END
        ._line 11
        exitd
```

附图 B.23　　引用 auto 区位字段的范例

当引用 auto 区内的一个位字段时，必须确定它位于可使用位操作指令来引用的范围内（在 FB 寄存器值的 32 字节内）。

附录 B.2.3　　指定 register 变量的寄存器名称

存储类的 auto 与 register 变量 （包括参数）可通过编译器映射到寄存器。

在 asm 函数中使用映射到寄存器的变量可通过编写如附图 B.24 中所示的程序。[2]

```
asm( "   优化代码      $$ " , 变量名称 );
```

附图 B.24　　register 变量的描述格式

2. 若需要使用寄存器修饰符将变量强制映射到寄存器，则在编译时指定 – fenable_register (-fER) 的选项。

只能使用这个格式来指定两个变量名称。附图 B.25 显示引用 register 变量及编译结果的范例。

- C 源文件：

```
void    func(void)
{
        register int    i=1;            ← 变量 “i” 是一个 register 变量

        asm("           mov.w           $$,A1",i);
}
```

- 汇编语言源文件 （编译结果）：

```
;## # FUNCTION func
;## # ARG Size(0) Auto Size(0) Context Size(4)
        .section        program,CODE,ALIGN
        ._file          'reg.c'
        .align
        ._line 3
;## # C_SRC : {
        .glb            _func
_func:
        ._line 4
;## # C_SRC : register inti=1;
        mov.w           #0001H,R0 ; i
        ._line 6
;## # C_SRC : asm("  mov.w           $$,A1",i);
;#### ASM START
        mov.w           R0,A1           ← R0 寄存器转移到 A1 寄存器
;#### ASM END
```

附图 B.25　　引用 register 变量及其编译结果的范例

在 NC30 中，函数中所使用的 register 变量以动态方式被分配。在任何一个位置，用于 register 变量的寄存器并不需要一直是同一个寄存器。因此，在 asm 函数中直接指定的寄存器，可能在编译后导致不同的运行结果。因此，我们建议您使用此函数来检查 register 变量。

附录 B.2.4　　指定 extern 和 static 变量的符号名称

以 C 编写的 extern 和 static 存储类变量将作为符号进行引用。

可以使用附图 B.26 中所示的格式来在 asm 函数中使用 extern 及 static 变量。

```
asm( "   优化代码       R1, $ " , 变量名称 );
```

附图 B.26　　指定符号名称的描述格式

使用此描述格式，只能指定两个变量名称。下面是所支持的变量名称类型：

- 变量名称
- 数组名称 [整数]
- 结构名称，成员名称 （不包括位字段成员）

```
 int      idata;
int       a[3];
struct TAG{
          int             i;
          int             k;
} s;

void      func(void)
{
              :
          asm("           MOV.W       R0, $$", idata);
              :
          asm("           MOV.W       R0, $$", a[2]);
              :
          asm("           MOV.W       R0, $$", s.i);
          （其余省略）
              :
}
```

附图 B.27　　指定的描述范例

参阅附图 B.28 中对 extern 和 static 变量进行引用的范例。

```
•  C 源文件：
extern int      ext_val;        ← extern 变量

void            func(void)
{
                static int      s_val;          ← static 变量

                asm("           mov.w       #01H,$$",ext_val);
                asm("           mov.w       #01H,$$",s_val);
}


•  汇编语言源文件 （编译结果）：
_func:
          ._line 7
;## # C_SRC : asm("          mov.w       #01H,$$",ext_val);
;#### ASM START
          mov.w             #01H,_ext_val               ← 移到 _ext_val
          ._line 8
;## # C_SRC : asm("          mov.w       #01H,$$",s_val);
          mov.w             #01H,___S0_s_val            ← 移到 __S0_e_val
;#### ASM END
          ._line 9
;## # C_SRC : }
          rts
E1:
          .glb              _ext_val
          .section          bss_NE,DATA
___S0_s_val: ;### C 的名称是 s_val
          .blkb 2
          .END
```

附图 B.28　　引用 extern 及 static 变量的范例

可以使用附图 B.29 中所示的格式来在 asm 函数中使用 extern 及 static 变量的 1 位位字段。（ 无法操作大于 2 位的位字段。

```
asm( "   优化代码  $b[ FB ]" , 位字段名称 );
```

附图 B.29　　指定符号名称的格式

可以使用这个格式来指定一个变量名称。请参阅附图 B.30 中的范例。

```
struct TAG{
        char        bit0:1;
        char        bit1:1;
        char        bit2:1;
        char        bit3:1;
} s;

void    func(void)
{
        asm("       bset        $b",s.bit1);
}
```

附图 B.30　　指定符号位的位置的范例

显示编译附图 B.31 中所示的 C 源文件的结果。

```
;## # FUNCTION func
;## # ARG Size(0) Auto Size(0) Context Size(4)
        .section    program,CODE,ALIGN
        ._file      'kk.c'
        .align
        ._line 10
;## # C_SRC : {
        .glb        _func
_func:
                    ._line 11
;## # C_SRC : asm("bset      $b",s.bit1);
;#### ASM START
        bset        1,_s    ← 引用结构 s 的位字段 bit0
;#### ASM END
        ._line 12
;## # C_SRC : }
        rts
E1:
        .section    bss_NO,DATA
        .glb        _s
_s:
        .blkb 1
        .END
```

附图 B.31　　引用符号位字段的范例

当引用 extern 或 static 变量的位字段时，必须确定它们位于可使用位操作指令直接进行引用的范围内（0000H 和 1FFFH 内）。

附录 B.2.5 不依赖存储类的指定

以 C 语言编写的变量可在 asm 函数中使用，而不须依赖该变量的存储类 （auto、 register[3]、 extern 或 static 变量）。

因此，可以通过附图 B.32 中所示的格式编写，在 asm 函数中使用任何以 C 语言编写的变量。[4]

```
asm("    优化代码      R0, $@", 变量名称 );
```

附图 B.32 不依赖变量的存储类的描述格式

只能使用这个格式来指定一个变量名称。附图 B.33 显示引用变量及编译结果的范例。

- C 源文件：

```
extern int              e_val;                          ← extern 变量

void    func(void)
{
        int             f_val;                          ← auto 变量
        register int    r_val;                          ← register 变量
        static int      s_val;                          ← static 变量

        asm("           mov.w       #1, $@", e_val);    ← 引用 extern 变量
        asm("           mov.w       #2, $@", f_val);    ← 引用 auto 变量
        asm("           mov.w       #3, $@", r_val);    ← 引用 register 变量
        asm("           mov.w       #4, $@", s_val);    ← 引用 static 变量
        asm("           mov.w       $@, $@", f_val,r_val);
}
```

- 汇编语言源文件 （编译结果）

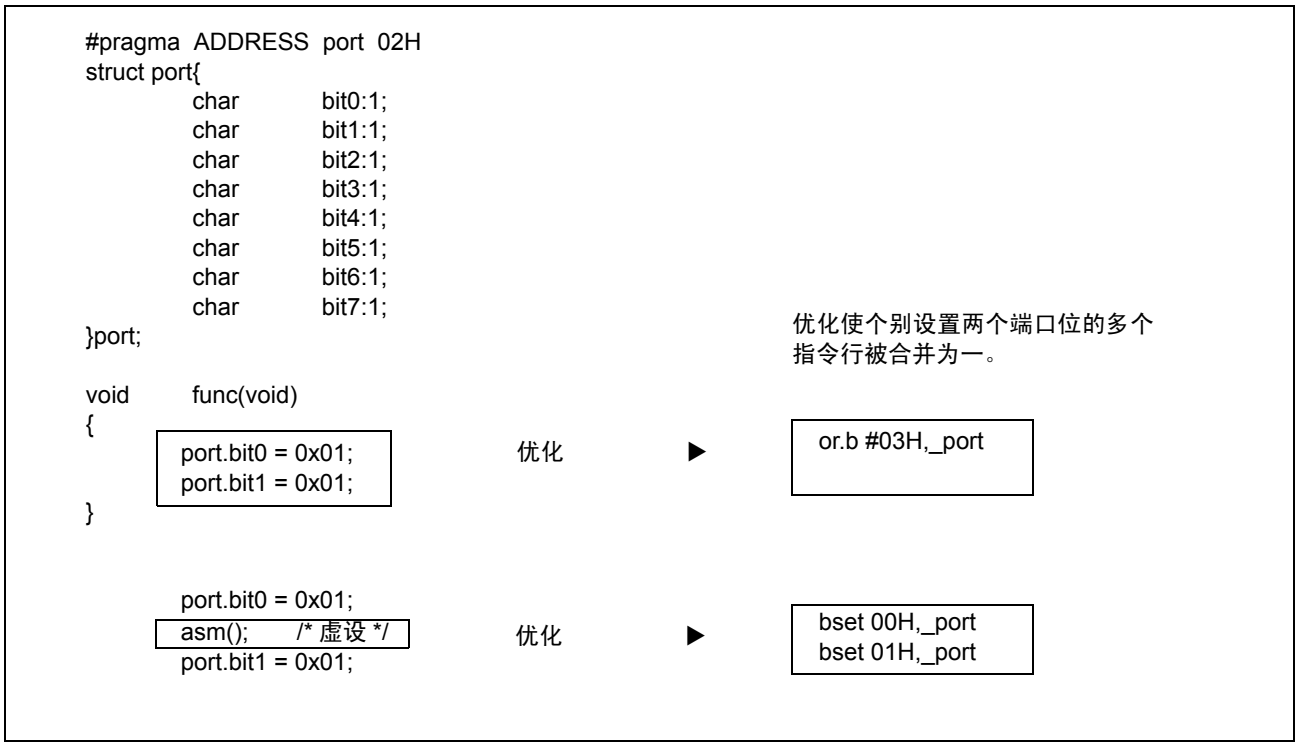
```
        .glb            _func
_func:
        enter           #02H
        pushm           R1
        ._line 9
;## # C_SRC : asm("     mov.w       #1, $@", e_val);
;#### ASM START
        mov.w           #1, _e_val:16                   ← 引用 extern 变量
        ._line 10
;## # C_SRC : asm("     mov.w       #2, $@", f_val);
        mov.w           #2, -2[FB]                      ← 引用 auto 变量
        ._line 11
;## # C_SRC : asm("     mov.w       #3, $@", r_val);
        mov.w           #3, R1                          ← 引用 register 变量
        ._line 12
;## # C_SRC : asm("     mov.w       #4, $@", s_val);
        mov.w           #4, ___S0_s_val:16              ← 引用 static 变量
        ._line 13
;## # C_SRC : asm("     mov.w       $@, $@", f_val,r_val);
        mov.w           -2[FB], R1
;#### ASM END
```

附图 B.33 引用各个存储类的变量的范例

3. 即使指定为寄存器限定，它也不会被限制分配给寄存器。
4. 不论它在编译时被分配的是哪一种存储类，都请您给予确定。

附录 B.2.6 选择性的禁止优化

在附图 B.34 中，虚设的 asm 函数被用来选择性的禁止一部分优化。



附图 B.34 使用虚设 asm 来禁止优化的范例

附录 B.2.7 有关 asm 函数的注意事项

(a) 有关 asm 函数的扩展功能

当在下列处理中使用 asm 函数时，确保使用编写范例中所显示的格式。

(1) 请勿使用帧基址寄存器 (FB) 的偏移来指定 auto 变量或参数，或 1 位的位字段。使用附图 B.35 中所显示的格式来指定 auto 变量和参数。

```
asm("        MOV.W    #01H,$$[FB]", i);    ← 引用 auto 变量的格式
asm("        BSET     $$[FB]", s.bit0);    ← 检查 auto 位字段的格式
```

附图 B.35 编写 asm 函数的范例 (1/2)

(2) 可以在 NC30 中指定 register 存储类。在用 -fenable_register (-fER) 选项来编译 register 类变量时， asm 函数中的 register 变量使用附图 B.36 中所显示的格式。

```
asm("        MOV.W    #0,$$", i);          ← 检查 register 变量的格式
```

附图 B.36 编写 asm 函数的范例 (2/2)

请注意当指定 -O[1-5]、-OR、-OS、-OR_MAX 或 -OS_MAX 的选项时，通过 register 传递的参数可能会作为 register 变量处理，而不会移到 auto 区，以便增进代码效率。在这种情况下，当在 asm 函数中指定参数时，汇编语言将使用寄存器名称，而不是变量的 FB 偏移来输出。

(3) 当引用 asm 函数中的参数时

编译器会在变量（包括参数和 auto 变量）有效的处理程序间隔中分析程序流程。基于这个原因，若参数或 auto 变量在 asm 函数中被直接引用，这些有效间隔的管理将被破坏，而编译器将无法正确输出代码。

因此，为了在编写的 asm 函数中引用参数或 auto 变量，总是要确保使用 asm 函数的 “$$, $b, $@” 功能。

```
void    func( int i,int j)
{
        asm ("          mov.w          2[FB],4[FB]");                  /* j = i; */
}
```

附图 B.37　　无法正确引用的范例

在上述情况中，由于编译器确定 “i” 和 “j” 未在函数 func 中使用，它不会输出构成引用参数的帧所需的代码。基于这个原因，参数将无法被正确引用。

(4) 关于在 asm 函数内的转移

编译器会在寄存器与变量分别有效的处理程序间隔中分析程序流程。请勿在 asm 函数中编写将影响程序流程的转移语句（包括条件转移）。

(b) 关于寄存器

- 请勿破坏 asm 函数中的寄存器。如果寄存器的值被销毁，则使用 push 和 pop 指令以保存及恢复寄存器。
- NC30 设置了在启动程序初始化后以固定模式使用 SB 寄存器的条件属性。若修改了 SB 寄存器，则必须如附图 B.38 所示般编写语句，以在连续的 asm 函数执行结束后将它恢复。

```
asm("       .SB       0);
asm("       LDC       #0H, SB");          ← SB 已更改
asm("       MOV.W     R0, _port[SB]");
              :
            （已省略）
              :
asm("       .SB       __SB__);
asm("       LDC       #__SB__,SB");       ← SB 返回到原始状态
```

附图 B.38 恢复已修改的静态基址 (SB) 寄存器

- 由于 FB 寄存器被用于堆栈帧指针，因此请勿通过 asm 函数来修改它。

(c) 有关标签的注意事项

由 NC30 所生成的汇编器源文件具有如附图 B.39 中所示的内部标签格式。因此，应该避免在 asm 函数中使用可能造成名称重复的标签。

```
• 具有一个大写字母及一个或多个数字的标签
范例：  A1:
        C9830:


• 具有两个或多个字符及下划线 （_）的前缀的标签
范例：  __LABEL:
        ___START:
```

附图 B.39 禁止在 asm 函数中使用的标签格式

附录 B.3 日文字符的描述

NC30 允许您将日文字符包含在 C 源程序中。本章将说明有关方法。

附录 B.3.1 日文字符的概述

相对于使用一个字节的字母和其它字符，日文字符需要使用两个字节。NC30 允许在字符串、字符常数和注释中使用这种 2 个字节的字符。下列字符类型可被包含在内：

- 日文汉字
- 平假名
- 全角片假名
- 半角片假名

只有下列日文汉字代码系统可在 NC30 中用于日文字符。

- EUC （除了由 3 个字节的代码组成的用户定义字符）
- Shift JIS （SJIS）

附录 B.3.2 使用日文字符所需的设置

若要使用日文汉字代码，必须先设置下列环境变量。默认指定：

- 环境变量指定输入代码系统.....NCKIN
- 环境变量指定输出代码系统.....NCKOUT

附图 B.40 是设置环境变量的一个范例。

将下列设置包括在 autoexec.bat 文件中：

```
设置 NCKIN=SJIS
设置 NCKOUT=SJIS
```

附图 B.40 设置环境变量 NCKIN 和 NCKOUT 的范例

在 NC30 中，输入日文汉字代码由 cpp30 预处理器进行处理。 cpp30 会将代码更改为 EUC 代码。在 ccom30 编译器标志分析的最后阶段中， EUC 代码将会被转换，以按照环境变量中的指定进行输出。

附录 B.3.3 字符串中的日文字符

附图 B.41 中显示在字符串中包括日文字符的格式。

```
L"漢字文字列"
```

附图 B.41 字符串中的日文汉字代码描述格式

若您使用一般字符串的格式来编写日文，在操作字符串时，它将被当作 char 类型的指针类型进行处理。因此，无法将它们作为 2 字节字符来进行操作。

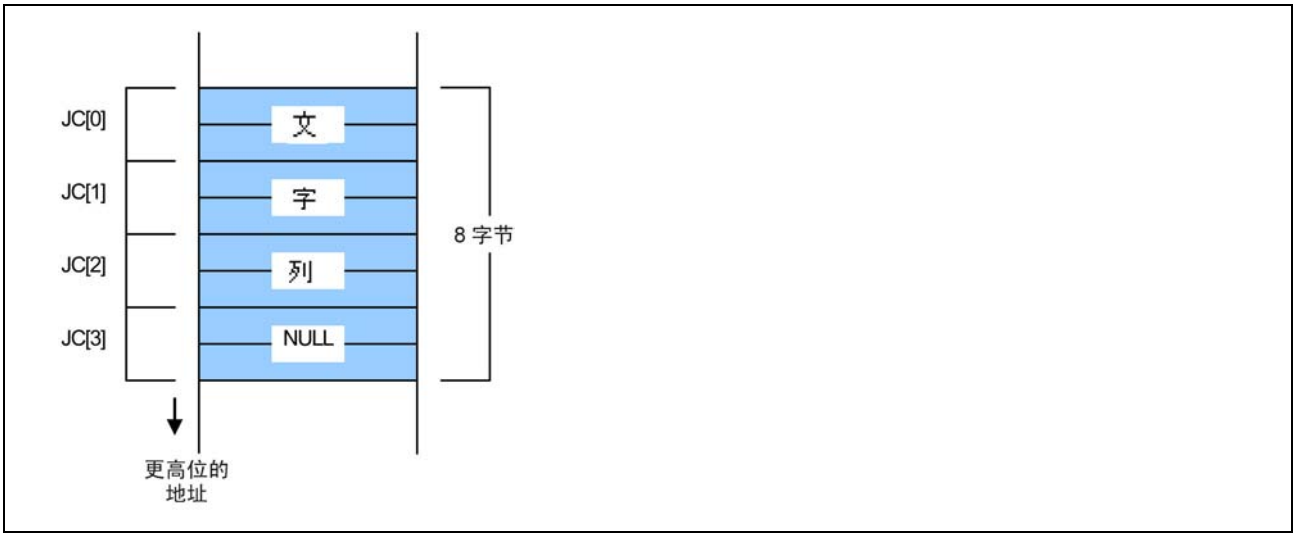
若要将日文作为 2 字节的字符进行处理，则必须在字符串前加上 L，以将它作为 wchar_t 类型的指针类型来处理。 wchar_t 类型在标准标题文件 stdlib.h 中被定义 （typedef）为 unsigned short 类型。

附图 B.42 中显示一个日文字符串的范例。

```
#include <stdlib.h>

void    func(void)
{
        wchar_t     JC[4] = L" 文字列";              ← [1]

        （其余省略）
          :
```

附图 B.42 日文字符串的描述范例

附图 B.43 是在附图 B.42 的 (1) 中初始化的字符串的存储器映像。



附图 B.43 wchar_t 类型字符串的存储器位置

附录 B.3.4 将日文字符用作字符常数

附图 B.44 中显示将日文字符用作字符常数的格式。

```
L' 漢 '
```

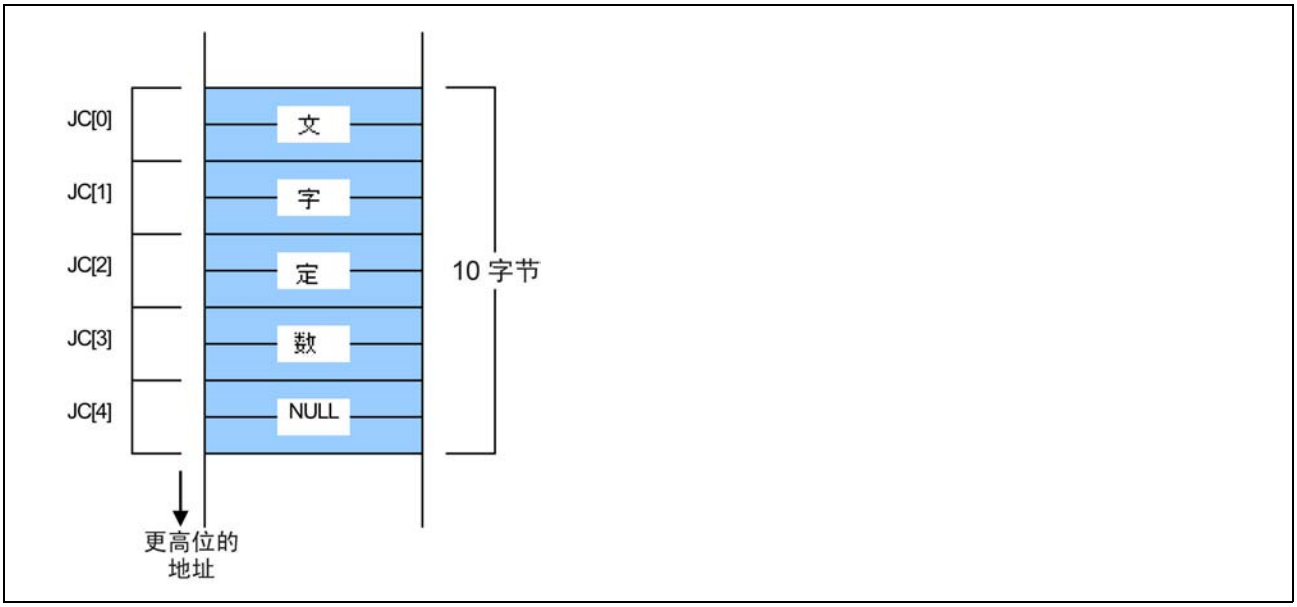
附图 B.44 字符串中的日文汉字代码描述格式

和字符串一样，在字符常数前加上 L，将它作为 wchar_t 类型处理。如果使用两个或多个字符作为字符常数，比如说，‘文字’，则只有第一个字符 “文”会变成字符常数。附图 B.45 中显示如何编写日文字符常数的范例。

```
#include <stdlib.h>

void    func(void)
{
        wchar_tJC[5];

        JC[0] = L' 文 ';
        JC[1] = L' 字 ';
        JC[2] = L' 定 ';
        JC[3] = L' 数 ';

        （其余省略）
          :
```

附图 B.45 日文汉字字符常数的描述格式

附图 B.46 是附图 B.45 中的字符常数所分配数组的存储器映像。



附图 B.46 wchar_t 类型字符常数所分配数组的存储器位置

附录 B.4 函数的默认参数声明

NC30 可让您使用与 C++ 功能相同的方式，来定义函数的参数默认值。本章将说明 NC30 声明函数的默认参数的功能。

附录 B.4.1 函数的默认参数声明的概述

NC30 可让您在声明函数原型时，通过指定参数默认值来使用隐含参数。通过这项功能，您将能节省编写调用函数时的常用值所需的时间和精力。

附录 B.4.2 函数的默认参数声明的格式

附图 B.47 中显示用来声明函数的默认参数的格式。

```
存储类说明符 Δ 类型声明符 Δ 声明符 ([ 虚设参数 [= 默认值或变量 ],...]);
```

附图 B.47 函数的默认参数的声明格式

附图 B.48 中显示一个函数声明的范例，而附图 B.49 中则显示附图 B.48 所显示的样品程序的编译结果。

```
int     func( int i=1 , int j=2 );      ← 将函数 func 中的默认参数值
                                          声明为第一个参数：1 及第二个参数：2。

void    main(void)
{
        func();         ← 实际参数包含第一个函数：1 及第二个参数：2。
        func(3);        ← 实际参数包含第一个参数：3 及第二个参数：2。
        func(3.5);      ← 实际参数包含第一个参数：3 及第二个参数：5。
}
```

附图 B.48 函数的默认参数的声明范例

```
;## # C_SRC :        {
                .glb        _main
_main:
                ._line5
;## # C_SRC :                    func();
        mov.w       #0002H,R2           ← 第二个参数：2
        mov.w       #0001H,R1           ← 第一个参数：1
        jsr         $func
        ._line      6
;## # C_SRC :                    func(3);
        mov.w       #0002H,R2           ← 第二个参数：2
        mov.w       #0003H,R1           ← 第一个参数：3
        jsr         $func
        ._line      7
;## # C_SRC :                    func(3.5);
        mov.w       #0005H,R2           ← 第二个参数：5
        mov.w       #0003H,R1           ← 第一个参数：3
        jsr         $func
        ._line      8
;## # C_SRC :        }
        rts
              :
        （已省略）
              :
```

注意：在 NC30 中，参数会从在函数中最后声明的参数开始，以相反顺序保存到堆栈。在此范例中，参数在处理过程中通过寄存器传递。

附图 B.49　　smp1.c （smp1.a30）的编译结果

可为函数的参数编写变量。附图 B.50 中显示以变量指定默认参数的范例。附图 B.51 则显示在附图 B.50 中所显示的样品程序的编译结果。

```
int     near sym ;
int     func( int i = sym);             ← 用变量来指定默认参数。

void    main(void)
{
        func();                         ← 将变量 （sym）用作参数来对函数进行调用。
}
              :
        （已省略）
              :
```

附图 B.50　　用变量指定默认参数的范例 （smp2.c）

```
_main:
        ._line 6
        mov.w       _sym,R1             ← 将变量 （sym）用作参数来对函数进行调用。
        jsr         $func
        ._line 7
        rts
```

附图 B.51　　smp2.c （smp2.a30）的编译结果

附录 B.4.3 声明函数的默认参数的限制

声明函数的默认参数具有下列限制。必须遵守这些限制。

(a) 当为多个参数指定默认值时

当在具有多个参数的函数中指定默认值时，总是确保从最后一个参数的值开始写入。附图 B.52 中显示了错误编写的范例。

```
void    func1(int i, int j=1, int k=2);          /* 正确 */
void    func2(int i, int j, int k=2);            /* 正确 */
void    func3(int i = 0, int j, int k);          /* 不正确 */
void    func4(int i = 0, int j, int k = 1);      /* 不正确 */
```

附图 B.52 原型声明的范例

(b) 当为默认值指定变量时

当为默认值指定变量时，请在声明您所指定的变量后，为函数编写原型声明。若为参数的默认值指定的变量未在声明函数的原型之前进行声明，处理将出现错误。

附录 B.5 直接插入函数声明

NC30 允许您用和 C++ 类似的方式来指定直接插入存储类。通过为函数指定直接插入存储类，可以进行直接插入扩展函数。本章将描述直接插入存储类的指定。

附录 B.5.1 直接插入存储类的概述

直接插入存储类说明符声明所指定的函数是要进行直接插入扩展的函数。直接插入存储类说明符向函数指出，使用它来声明的函数将被直接插入扩展。指定为直接插入存储类的函数具有直接嵌入的汇编级代码。

附录 B.5.2 直接插入存储类的声明格式

当声明直接插入存储类时，必须使用与 static 及 extern 类型的存储类说明符类似的语法格式来编写直接插入存储类说明符。附图 B.53 中显示用来声明直接插入存储类的格式。

```
inline 类型说明符函数 ;
```

附图 B.53 直接插入存储类的声明格式

附图 B.54 中显示函数的声明范例。

```
 inline int      func(int i)          ← 函数的原型声明
{
                 return i++;
}

 void            main(void)
 {
                 int      s;

                 s = func(s);         ← 函数主体的定义
}
```

附图 B.54 直接插入存储类的声明范例

```
                .SECTION    program,CODE,ALIGN
                ._file      'sample.c'
                .align
                ._line      7
        ;## # C_SRC :       {
                .glb        _main
        _main:
                enter       #02H
                pushm       R1
                ._line      10
        ;## # C_SRC :                   s = func(s);
                mov.w       -2[FB],R1   ;  s
                ._line      3
        ;## # C_SRC :                   return i++;          ← 直接插入存储类具有直接
                mov.w       R0,R1                               嵌入的代码
                add.w       #0001H,R1
                ._line      10
        ;## # C_SRC :                   s = func(s);
                mov.w       R0,-2[FB]   ;  s
                ._line      11
        ;## # C_SRC :       }
                popm        R1
                exitd
        E1:
                .END
```

附图 B.55　　样品程序 （smp.a30）的编译结果

附录 B.5.3　　直接插入存储类的限制

当指定直接插入存储类时，请注意下列事项：

(1)　有关直接插入函数的参数

直接插入函数的参数不可被 “structure”及 “union”使用。否则它将变成一项编译错误。

(2)　有关直接插入函数的间接调用

不可进行直接插入函数的间接调用。一旦描述了间接调用，它将变成一项编译错误。

(3)　有关直接插入函数的递归调用

不可进行直接插入函数的递归调用。一旦描述了递归调用，它将变成一项编译错误。

(4)　有关直接插入函数的定义

当为函数指定直接插入存储类时，确保在调用函数前先定义函数的主体。确保将该主体定义和函数编写在相同文件中。附图 B.56 中的描述在 NC30 中导致处理错误。

```
inline void     func(int i);

void            main(void)
{
                func(1);
}
```

[错误信息]
[Error(ccom):sample.c,line 5] inline function's body is not declared previously （之前未声明直接插入函数的主体）
===> func(1);
Sorry, compilation terminated because of these errors in main(). （对不起，由于 main() 中的这些错误，编译已终止。）

附图 B.56　　直接插入函数的错误编码范例 (1)

此外，当一些函数被用作普通函数之后，若再将该函数定义为直接插入函数， NC30 将出现错误。（请参阅附图 B.57。）

```
int             func(int i);

void            main(void)
{
                func(1);
}

inline int      func(int i)
{
                return i;
}
```

[错误信息]
[Error(ccom):in.c,line 9] inline function is called as normal function before （直接插入函数之前被作为普通函数调用）
===>{

附图 B.57　　直接插入函数的错误编码范例 (2)

(5)　有关直接插入函数的地址

直接插入函数本身并没有地址。因此，若为直接插入函数使用 & 运算符，软件将假定这是一项错误。

```
 inline int         func(int i)
{
                   return i;
}

void               main(void)
{
                   int              (*f)(int);

                   f = &func;
}
```

[错误信息]
[Error(ccom):sample.c,line 10] can't get inline function's address by '&' operator （无法通过 '&' 运算符获得直接插入函数的地址）
===> f = &func;
Sorry, compilation terminated because of these errors in main(). （对不起，由于 main() 中的这些错误，编译已终止。）

附图 B.58　　直接插入函数的错误编码范例 (3)

(6)　static 数据的声明

如果在直接插入函数中声明 static 数据，所声明的 static 数据的主体将被保留在文件中。基于这个原因，若直接插入函数包含两个或多个文件，将会造成对不同区的存取。因此，若在直接插入函数中使用 static 数据，请在函数之外进行声明。若在直接插入函数中发现 static 声明，NC30 将发出警告。瑞萨不建议在直接插入函数中编写 static 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 （直接插入函数中的 static 值）
===>static inti   = 0;

附图 B.59　　直接插入函数的错误编码范例 (4)

(7)　有关调试信息

NC30 不会输出直接插入函数的 C 语言级调试信息。因此，必须在汇编语言级调试直接插入函数。

附录 B.6 注释的扩展

NC30 允许使用 “/*” 和 “*/” 括起来的注释，及类似 C++ 的以 “//” 开头的注释。

附录 B.6.1 “//” 注释的概述

在 C 中，注释必须编写在 “/*” 和 “*/” 之间。而在 C++ 中，注释编写在 “//” 之后。

附录 B.6.2 “//” 的注释格式

行中包含 “//” 时，出现在 “//” 之后的任何文字将作为注释处理。

附图 B.60 中显示注释的格式。

```
// 注释
```

附图 B.60 注释的格式

附图 B.61 中显示注释的范例。

```
 void    func(void)
{
        int i;          /* 此为注释 *//
        int j;          // 此为注释
            :
        （已省略）
            :
}
```

附图 B.61 注释的范例

附录 B.6.3 “//” 和 “/*” 的优先级

“//” 和 “/*” 的优先级根据它们出现的顺序来决定。

因此，在新行代码中的 “//” 之间编写 “/*” 不会表示注释的开始。同时，在 “/*” 和 “*/” 之间编写 “//” 也不会表示注释的开始。

附录 B.7 #pragma 扩展功能

附录 B.7.1 #pragma 扩展功能的索引

下列索引表显示 #pragma[5] 扩展功能的内容及组成。

(a) 使用存储器映像扩展功能

附表 B.4 存储器映像扩展功能

扩展功能	描述
#pragma ROM	将指定的变量映射到 rom 语法：#pragma ROM △变量名称 范例：#pragma ROM val • 提供此功能的目的是保持与 NC77 和 NC79 的兼容性。 • 通过使用 const 修饰符，使该变量位于 rom 段中。
#pragma BIT	将外部变量所在的区声明为可在 16 位绝对寻址模式中使用 1 位操作指令的区（即存在于 00000H 到 01FFFH 地址中的变量）。 语法：#pragma BIT △变量名称 范例：#pragma BIT bit_data
#pragma SBDATA	声明数据使用 SB 相对寻址。 语法：#pragma SBDATA △变量名称 范例：#pragma SECTION bss nonval_data
#pragma SECTION	更改由 NC30 生成的段名称 语法：#pragma SECTION △段名称△新段名称 范例：#pragma SECTION bss nonval_data
#pragma STRUCT	1. 禁止在指定的标签内紧缩结构 语法：#pragma STRUCT △ structure_tag △ unpack 范例：#pragma STRUCT TAG1 unpack 2. 使用指定的标签来安排结构成员，并首先映射大小为偶数的成员 语法：#pragma STRUCT △ structure_tag △ arrange 范例：#pragma STRUCT TAG1 arrange
#pragma EXT4MPTR	显示变量是一个存取 4M 字节 ROM 扩展空间的指针。 语法：#pragma EXT4MPTR △变量名称 范例：#pragma EXT4MPTR sym
_ext4mptr	显示变量是一个存取 4M 字节 ROM 扩展空间的指针。 语法：_ext4mptr △变量名称 范例：_ext4mptr sym

5. 在之前的版本中，在 #pragma 之后指定指令函数的字 （如 ADDRESS、 INTERRUPT、 ASM 等）（简称子命令）必须大写。在这个版本中，子命令将不区分大小写，即无论大小写其效果都一样。

(b) 为目标器件使用扩展功能

附表 B.5 用于目标器件的扩展功能 (1)

扩展功能	描述
#pragma ADDRESS	指定变量的绝对地址。对于 near 变量，这是对存储体内地址的指定。 语法：#pragma ADDRESS △变量名称△绝对地址 范例：#pragma ADDRESS port0 2H
#pragma BITADDRESS	变量被分配到绝对地址所指定的位。 语法：#pragma BITADDRESS △变量名称△位的位置，绝对地址 范例：#pragma BITADDRESS io 1,100H
#pragma INTCALL	声明在软件中断中调用的以汇编器编写的函数 （int 指令）。 通过指定切换 [/c]，将能在调用函数时，并在进入该函数前，生成代码以将寄存器堆栈。（仅限于 NC308WA） 语法：#pragma INTCALL △ [/C] △ INT 编号△函数名称 （寄存器名称） 范例：#pragma INTCALL 25 func(R0, R1) 范例：#pragma INTCALL /C 25 func(R0, R1) 语法：#pragma INTCALL △ INT 编号△函数名称 () 范例：#pragma INTCALL 25 func() 范例：#pragma INTCALL /C 25 func() • 总是确保在编写此声明之前，先声明函数的原型。
#pragma INTERRUPT	声明用 C 语言编写的中断处理函数。这项声明使代码执行在进入或退出函数时生成的中断处理函数。此外，通过指定切换 /B，在调用函数时，将能切换寄存器到备用寄存器，而不是将它保存到堆栈。 语法： #pragma INTERRUPT △ [/B|/E|/V] △中断处理函数名称 #pragma INTERRUPT △ [/B|/E] △中断向量编号△中断处理函数名称 范例： #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 /V int_func ()

附表 B.6　　用于目标器件的扩展功能 (2)

扩展功能	描述
#pragma PARAMETER	声明当调用汇编器函数时，参数将通过指定的寄存器传递。 通过指定切换 [/c]，将能在调用函数时，并在进入该函数前，生成代码以将寄存器堆栈。（仅限于 NC308WA） 语法：#pragma PARAMETER △ [/C] △函数名称 (寄存器名称) 范例：#pragma PARAMETER asm_func(R0,R1) 范例：#pragma PARAMETER /C asm_func(R0,R1) • 总是确保在编写此声明之前，先声明函数的原型。
#pragma SPECIAL	声明 special 页子例程调用函数 语法： #pragma SPECIAL△编号 . 函数名称() #pragma SPECIAL△函数名称(vect= 编号) 范例： #pragma SPECIAL 30 func() #pragma SPECIAL func() (vect=30)

(c)　使用 MR30 扩展功能

附表 B.7　　用于 MR30 的扩展功能

扩展功能	描述
#pragma ALMHANDLER	声明用于 M16C 系列警报处理函数的实时操作系统名称 语法：#pragma ALMHANDLER △函数名称 范例：#pragma ALMHANDLER alm_func
#pragma CYCHANDLER	声明用于 M16C 系列起始周期处理函数的实时操作系统名称 语法：#pragma CYCHANDLER △函数名称 范例：#pragma CYCHANDLER cyc_func
#pragma INTHANDLER #pragma HANDLER	声明用于 M16C 系列中断处理函数的实时操作系统名称 语法 1：#pragma INTHANDLER △函数名称 #pragma INTHANDLER△[/E]△函数名称 语法 2：#pragma HANDLER △函数名称 #pragma HANDLER △ [/E] △函数名称 范例：　#pragma INTHANDLER int_func
#pragma TASK	声明用于 M16C 系列任务开始函数的实时操作系统名称 语法：#pragma TASK △任务开始函数名称 范例：#pragma TASK task1

补充：上述扩展功能通常由配置器生成，因此用户不须加以理会。

(d)　其它扩展功能

附表 B.8　　使用直接插入汇编器描述函数

扩展功能	描述
#pragma ASM #pragma ENDASM	为以汇编语言编写的语句指定区。 语法：#pragma △ ASM #pragma △ ENDASM 范例：#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 范例：#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 范例：#pragma JSRW func
#pragma PAGE	在汇编器列表文件中表示新页的位置。 语法：#pragma △ PAGE 范例：#pragma PAGE
#pragma __ASMMACRO	声明由汇编器宏定义的函数。 语法：#pragma __ASMMACRO △函数名称 (寄存器名称 , ...) 范例：#pragma __ASMMACRO mul(R0,R1)

附录 B.7.2 使用存储器映像扩展功能

NC30 具有下列存储器映像扩展功能。

#pragma ROM

映射到 rom 段

功能： 将特定数据 （变量）映射到 rom 段

语法： #pragma ROM △变量名称

描述： 此扩展功能仅对符合下列其中一项条件的变量有效：

- 在函数以外定义的非 extern 变量 （具有保留区的变量）
- 在函数内声明为 static 的变量

规则：

1. 若指定的不是变量，该指定将被忽略。
2. 若指定 #pragma ROM 超过一次，并不会发生错误。
3. 若不包含初始化表达式，映射到 rom 段的数据将具有 0 的初始值。

范例：

```
[C 语言源程序 ]

#pragma     ROM  i
unsigned int                i;          ← 变量 i 符合条件 [1]

void        func(void)
{
            static int      i = 20;     ← 变量 i 符合条件 [2]
                :
            （其余省略）


[ 汇编语言源程序 ]

            .SECTIONrom_NE,ROMDATA
___S0_i: ;### C 的名称是 i              ← 变量 i 符合条件 [2]
            .word       0014H
            .glb    _i
_i:                                     ← 变量 i 符合条件 [1]
            .byte       00H
            .byte       00H
```

附图 B.62 使用 #pragma ROM 声明的范例

注意： 提供此功能的目的是保持与 NC77 和 NC79 的兼容性。通过使用 const 修饰符，使该变量位于 rom 段中。

#pragma BIT

使用变量描述函数的 SB 相对寻址

功能： 声明外部变量所在的区为可在 16 位绝对寻址模式中使用一位操作指令的区。

语法： #pragma BIT △ 变量名称

描述： M16C/60 系列可让您以可提高 ROM 效率的 16 位绝对寻址模式，对位于 00000H 到 01FFFH 地址区中的 extern 变量使用一位操作指令。
通过 #pragma BIT 声明的变量，被假设位于可直接对其使用一位操作指令的区中。

规则：
1. 若 #pragma BIT 不是对 extern 变量使用，它将被忽略且不起任何作用。
2. 当在 #pragma BIT 中声明具有 1 位宽度的 extern 变量后，将总是直接输出 1 位指令。因此，当包含 #pragma BIT 声明时，应确保变量被映射到 0 和 01FFFH 之间。

范例：

```
#pragma BIT bit_data
struct bit_data{
                char bit0:1;
                char bit1:1;
                char bit2:1;
                char bit3:1;
                char bit4:1;
                char bit5:1;
                char bit6:1;
                char bit7:1;
}bit_data;
func( void )
{
                bit_data.bit1 = 0;
```

附图 B.63　使用 #pragma BIT 声明的范例

注意： 1 位指令会在下列任何一项条件下生成：
1. 指定了 -fbit(-fB) 选项，且目标对象是 near 类型的变量
2. 目标对象是通过 #pragma SBDATA 声明的变量
3. 目标对象是通过 #pragma ADDRESS 声明的变量，且变量位于 0000H 到 01FFFH 之间的地址
4. 目标对象是通过 #pragma BIT 声明的变量
5. 变量映射到 FB 寄存器值 32 字节内的区

#pragma SBDATA

使用变量描述函数的 SB 相对寻址

功能： 声明数据使用 SB 相对寻址。

语法： #pragma SBDATA △变量名称

描述： M16C/60 系列可选择能使用 SB 相对寻址来有效执行的指令。在引用数据时，变量可使用 #pragma SBDATA 声明 SB 相对寻址。这项功能会帮助生成可提高 ROM 效率的代码。

规则：

1. 被声明为 #pragma SBDATA 的变量，将由汇编器的伪指令 .SBSYM 进行声明。
2. 若 #pragma SBDATA 指定的不是变量，它将被忽略且不起任何作用。
3. 若所指定的变量在函数中被声明为 static 变量，则 #pragma SBDATA 的声明将被忽略且不起任何作用。
4. 当分配存储器时，声明为 #pragma SBDATA 的变量将被放置在 SBDATA 属性段中。
5. 不能够以相同的变量同时指定 #pragma SBDATA。
6. 若为 ROM 数据声明了 #pragma SBDATA，数据不会被放置在 SBDATA 属性段中。

范例：

```
#pragma  SBDATA  sym_data
struct sym_data{
                    char        bit0:1;
                    char        bit1:1;
                    char        bit2:1;
                    char        bit3:1;
                    char        bit4:1;
                    char        bit5:1;
                    char        bit6:1;
                    char        bit7:1;
}sym_data;

void                func(void)
{
                    sym_data.bit1 = 0;
                        :
                    （已省略）
                        :
```

附图 B.64　使用 #pragma SBDATA 声明的范例

注意： NC30 假设 SB 寄存器会在复位后初始化，然后用作固定量。

#pragma SECTION

更改段名称

功能： 更改由 NC30 生成的段名称

语法： #pragma SECTION △段名称△新段名称

描述： 在 #pragma SECTION 声明中指定程序段、数据段和 rom 段，将更改接下来所有函数中的段名称。在 #pragma SECTION 声明中指定 bss 段，将更改在该文件中定义的所有数据段的名称。若您在使用此功能更改段名称、初始化等等之后，需要添加或更改段名称，则须分别在每个段的启动程序中进行。

- 可以在一个文件中指定 “#pragma SECTION data”和 “#pragma section program”两次或多次。
- 所有其它段的名称将无法被更改两次或以上。

范例：

```
[C 源程序 ]

#pragma     SECTION    program     pro1          ← 将程序段的名称更改为 pro1
void        func( void );
                :
            （其余省略）

[ 汇编语言源程序 ]

;## # FUNCTION func
                .section    pro1                  ← 映射到 pro1 段
                ._file      'smp.c'
                ._line      9
                .glb        _func
_func:

[ 将数据段的名称从 data 更改为 data1]

#pragma     SECTION  data  data1
int         i;                                    ← 映射到 data1_NE 段

void        func(void)
{
            （其余省略）
}

#pragma     SECTION  data  data2
int         j;                                    ← 映射到 data2_NE 段 */

void        sub(void)
{
            （其余省略） }
}
```

附图 B.65　使用 #pragma SECTION 声明的范例

补充： 当修改段的名称时，请注意段的位置属性 （如 _NE 或 _NEI）会附加在段名称之后。

注意： 在本编译器 V.3.10 或之前的版本中，数据和 rom 段同 bss 段一样，只能在文件单位中更改名称。基于这个原因，使用 V.3.10 或之前版本建立的程序时，需要注意编写 #PRAGMA SECTION 的位置。字符串数据将以最后声明的 rom 段名称输出。

#pragma STRUCT

控制结构映像

功能：
1. 禁止结构的紧缩
2. 排列结构成员

语法：
1. #pragma STRUCT △ structure_tag △ unpack
2. #pragma STRUCT △ structure_tag △ arrange

描述与范例：
在 NC30 中，结构都经过紧缩。例如，附图 B.66 中的结构成员在没有任何填充的情形下按照它们声明的顺序编排。

```
struct s {
    int     i;
    char    c;
    int     j;
};
```

成员 名称	类型	大小	映像位置 （偏移）
i	int	16bits int	0
c	char	8bits char	2
j	int	16bits int	3

附图 B.66　结构成员的映射范例 (1)

规则：
1. 禁止紧缩
   这个 NC30 扩展功能允许您控制结构成员的映像。附图 B.67 是使用 #pragma STRUCT 来映射附图 B.66 中的结构成员以禁止紧缩的范例。

```
struct s {
    int     i;
    char    c;
    int     j;
};
```

成员名称	类型	大小	映像位置 （偏移）
i	int	16bits int	0
c	char	8bits char	2
j	int	16bits int	3
填充	(char)	8bits (char)	-

附图 B.67　结构成员的映像范例 (2)

如附图 B.67 所示，如果结构成员的大小是奇数字节， #pragma STRUCT 将会在最后一个成员后面添加 1 字节作为紧缩。因此，若您使用 #pragma STRUCT 来禁止填充，所有结构将具有偶数的字节大小。

#pragma STRUCT

控制结构映像

描述：

2. 排列成员

这个 NC30 扩展功能可让您先映射所有奇数大小的结构成员，然后再映射偶数大小的成员。附图 B.68 显示当使用 #pragma STRUCT 来排列附图 B.67 中的结构时的偏移。

```
struct s {
   int     i;
   char    c;
   int     j;
};
```

成员名称	类型	大小	映像位置 （偏移）
i	int	16bits int	0
j	int	16bits int	2
c	char	8bits char	4

附图 B.68　　结构成员的映像范例 (3)

在定义结构成员之前，您必须声明 #pragma STRUCT 来禁止紧缩及排列结构成员。

范例：

```
#pragma      STRUCT  TAG  unpack
struct TAG{
                int          i;
                char         c;
} s1;
```

附图 B.69　　#pragma STRUCT 声明的范例

#pragma EXT4MPTR

定义位于 4M 字节扩展空间 ROM 区中的数据

功能： 显示变量是一个存取 4M 字节 ROM 扩展空间的指针。

语法： #pragma EXT4MPTR △指针名称

描述： 这项功能是为在 M16C/62 群的一些产品中可用的扩展模式 2 （4M 字节扩展模式）提供。声明指针变量以存取 4M 字节空间。当进行这项声明时，编译器将生成代码，以切换到存取 4M 字节空间所需使用的存储体。
在首先使用指针的地方，每个函数都将生成这个存储体切换代码。在接下来的操作中，存储体只被设置一次。
当使用多个指针变量时，使用 “-fchange_bank_always (-fCBA)”选项，以在每次程序存取 4M 字节空间时设置存储体。

范例：

```
[C 源程序 ]
struct tagh{
        int bitmap;
        char code;
}far *pointer;
#pragma EXT4MPTR pointer
main()
{
        int data;
        data = pointer->bitmap;
}
        mov.w _pointer, A0
        mov.w _pointer+2, A1
        mov.w A1,__BankSelect          ← 更改存储体
        bclr 3,A1
        bset 2,A1
        lde.w [A1A0],-2[FB]
```

附图 B.70　使用 #pragma EXT4MPTR 声明的范例

注意：
1. 在使用这项功能之前，查看单片机和系统 （硬件）是否支持 4M 字节的扩展空间模式。
2. 若使用了 -R8C 选项，这项声明将被忽略。

_ext4mptr

定义位于 4M 字节扩展空间 ROM 区中的数据

功能： 显示变量是一个存取 4M 字节 ROM 扩展空间的指针。

语法： _ext4mptr far △指针名称

描述： 这项功能是为在 M16C/62 群的一些产品中可用的扩展模式 2 （4M 字节扩展模式）提供。声明用于存取 4M 字节空间的指针变量。当进行这项声明时，编译器将生成代码，以切换到存取 4M 字节空间所需使用的存储体。
在指针首先使用的地方，每个函数都将生成这个存储体切换代码。在接下来的操作中，存储体只被设置一次。
当使用多个指针变量时，使用 “-fchange_bank_always (-fCBA)”选项，以在每次程序存取 4M 字节空间时设置存储体。

范例：

```
[C 源程序 ]
struct tagh{
        int bitmap;
        char code;
};
struct tagh _ext4mptr *pointer;
main()
{
        int data;
        data = pointer->bitmap;
}
        mov.w _pointer,A0
        mov.w _pointer+2,A1
        mov.w A1,__BankSelect          ← 更改存储体
        bclr 3,A1
        bset 2,A1
        lde.w [A1A0],-2[FB]
```

附图 B.71　使用 #pragma EXT4MPTR 声明的范例

注意：
1. 在使用这项功能之前，查看单片机和系统 （硬件）是否支持 4M 字节的扩展空间模式。
2. 若使用了 -R8C 选项，这项声明将被忽略。

附录 B.7.3 为目标器件使用扩展功能

NC30 包含了用于目标器件的下列扩展功能。

#pragma ADDRESS

指定 I/O 变量的绝对地址

功能： 指定变量的绝对地址。对于 near 变量，所指定的地址位于存储体内。

语法： #pragma ADDRESS △变量名称△绝对地址

描述： 在这项声明中指定的绝对地址将在汇编器文件中扩展为字符串，并以伪指令 .EQU 进行定义。因此，编写数字值的格式将视汇编器而定，如下所示：

- 将 ‘B’ 或 ‘b’ 附加到二进制数字。
- 将 ‘O’ 或 ‘o’ 附加到八进制数字。
- 仅编写十进制整数。
- 将 ‘H’ 或 ‘h’ 附加到十六进制数字。若数字以字母 A 到 F 开头，则须在之前加上 0。

规则：

1. 在 #pragma ADDRESS 中指定的变量的所有存储类，如 extern 和 static，将无效。
2. 在 #pragma ADDRESS 中指定的变量，只有那些在函数以外定义的变量有效。
3. #pragma ADDRESS 只对之前声明的变量有效。
4. 若指定的不是变量， #pragma ADDRESS 将无效。
5. 重复声明 #pragma ADDRESS 不会发生错误，但只有最后声明的地址有效。
6. 若包含了初始化表达式，则会输出警告，并且该初始化表达式无效。
7. 一般情况下， #pragma ADDRESS 会在 I/O 变量上操作，因此即使未正确指定 volatile，编译器也会假设已指定 volatile 来进行处理。
8. 在 #pragma ADDRESS 声明中声明的变量，将无法进行外部引用。

范例：

```
#pragma     ADDRESS  port  24H
int         io;

void        func(void)
{
            io = 10;
}
```

附图 B.72 #pragma ADDRESS 声明

范例： 不过，当变量在 #pragma ADDRESS 的指定之前使用时， #pragma ADDRESS 的指定将无效，如下所示。

```
char        port;

void        func(void)
{
            port = 0;                /* 在指定 #pragma ADDRESS 之前使用变量 */
}

#pragma  ADDRESS  port  100H
```

附图 B.73 #pragma ADDRESS 的指定不发挥作用的情况

#pragma BITADDRESS

输入输出变量的位指定绝对地址分配功能

功能： 变量被分配到特定绝对地址所指定的位。

语法： #pragma BITADDRESS △变量名称△位的位置 , 绝对地址

描述： 在这项声明中指定的绝对地址将在汇编器文件中扩展为字符串，并以伪指令 .BITEQU 进行定义。因此，编写数字值的格式将视汇编器而定，如下所示：

1. 位的位置
   - 有效范围是 0-65535。只限十进制数字。
2. 地址
   - 将 ‘B’ 或 ‘b’ 附加到二进制数字。
   - 将 ‘O’ 或 ‘o’ 附加到八进制数字。
   - 仅编写十进制整数。
   - 将 ‘H’ 或 ‘h’ 附加到十六进制数字。若数字以字母 A 到 F 开头，则在之前加上 0。

规则：

1. 只有 _Bool 类型变量可被指定为变量名称。当指定的不是 _Bool 类型的变量时，则会发生错误。
2. 在 #pragma BITADDRESS 中指定的变量的所有存储类，如 extern 和 static，将无效。
3. 在 #pragma BITADDRESS 中指定的变量，只有那些在函数以外定义的变量有效。
4. #pragma BITADDRESS 只对之前声明的变量有效。
5. 若您指定的不是变量， #pragma BITADDRESS 将无效。
6. 重复声明 #pragma BITADDRESS 不会发生错误，但只有最后声明的地址有效。
7. 若您包含初始化表达式，则将发生错误。
8. 一般情况下， #pragma BITADDRESS 会在 I/O 变量上操作，因此即使未正确指定 volatile，编译器也会假设已指定 volatile 来进行处理。

范例：

```
#pragma  BITADDRESS    io    1,100H
_Bool      io;

void       func(void)
{
           io = 1;
}
```

附图 B.74　#pragma BITADDRESS 声明

#pragma INTCALL

声明由 INT 指令调用的函数

功能： 声明由软件中断 （由 int 指令）调用的函数

语法： #pragma INTCALL △ [/C] △ INT 编号△汇编器函数名称 (寄存器名称 , 寄存器名称 , ...)
#pragma INTCALL △ [/C] △ INT 编号△函数名称 ()

描述： 此扩展函数使用 INT 编号来声明由软件中断调用的汇编器函数。
当调用汇编器函数时，它的参数将通过寄存器传递。

- [/C]

  通过指定切换 [/c]，将能在调用函数时，并在进入该函数前，生成代码以将寄存器堆栈。（仅限于 NC308WA）

规则：

- 声明汇编器函数
  1. 在 #pragma INTCALL 声明之前，确保先包含汇编器函数的原型声明。若没有原型声明，则会输出警告，同时 #pragma INTCALL 的声明将被忽略。
  2. 在原型声明中须遵守下列事项：
     a. 确保原型声明中的参数数量与 #pragma INTCALL 声明中的数量一致。
     b. 无法在汇编器函数的参数中声明下列类型：
        - 结构类型和联合类型
        - double 类型
        - long long 类型

     c. 无法将下列函数声明为汇编器函数的返回值：
        - 返回结构或联合的函数
  3. 可以在调用时为参数使用下列寄存器：
     - float 类型， long 类型 （32 位寄存器）
       R2R0 和 R3R1
     - far 指针类型 （24 位寄存器）
       A0、 A1、 R2R0 和 R3R1
     - near 指针类型 （16 位寄存器）
       A0、 A1、 R0、 R1、 R2 和 R3
     - char 类型和 _Bool 类型 （8 位寄存器）
       R0L、 R0H、 R1L 和 R1H

     * 寄存器名称中无大小写字母区分。
  4. INT 编号只可以使用十进制。

- 声明主体以 C 编写的函数
  1. 在 #pragma INTCALL 声明之前，确保先包含原型声明。若没有原型声明，则会输出警告，同时 #pragma INTCALL 的声明将被忽略。
  2. 您将无法指定具有 #pragma INTCALL 声明的函数的参数寄存器名称。
  3. 在原型声明中须遵守下列事项：
     a. 如函数调用规则中所述，在原型声明中，可声明的函数只限于所有参数通过寄存器传递的函数。
     b. 无法将下列函数声明为函数的返回值：
        - 返回结构或联合的函数
  4. INT 编号只可以使用十进制。

#pragma INTCALL

声明由 INT 指令调用的函数

范例：

```
int       asm_func(unsigned long, unsigned int);     ← 汇编器函数的
#pragma   INTCALL  25  asm_func(R2R0, R1)            原型声明

void      main(void)
{
          int       i;
          long      l;

          i = 0x7FFD;
          l = 0x007F;

          asm_func( l, i );                          ← 调用汇编器函数
}
```

附图 B.75　#pragma INTCALL 声明 （asm 函数）的范例 (1)

```
int       c_func(unsigned int, unsigned int);   ← C 函数的原型声明
#pragma   INTCALL  25  c_func();                ← 不能指定寄存器。

void      main(void)
{
          int     i, j;

          i = 0x7FFD;
          j = 0x007F;

          c_func( i, j );                       ← 调用 C 函数
}
```

附图 B.76　#pragma INTCALL 声明 （C 语言函数）的范例 (2)

注意：　若要使用产品随附的启动文件，必须在使用前修改向量段的内容。有关如何修改的详情，请参考 “第 2 章准备启动程序”。

#pragma INTERRUPT

声明中断函数

功能： 声明中断处理程序

语法：

1. #pragma INTERRUPT △ [/B|/E|/V] △中断处理程序名称
2. #pragmaINTERRUPT △ [/B|/E] △中断向量编号△中断处理程序名称
3. #pragmaINTERRUPT △ [/B|/E] △中断处理程序名称 (vect= 中断向量编号)

描述：

1. 通过使用上述格式来声明以 C 编写的中断处理函数， NC30 将会生成代码，以在进入和退出函数时执行下列中断处理。
   - 在进入函数处理中，单片机的所有寄存器都将被保存到堆栈。
   - 在退出函数处理中，将恢复所保存的寄存器，并通过 REIT 指令将主控权返回给进行调用的函数。
2. 可以在这项声明中指定 /B 或 /E 或 /V：
   - [/B]
     ：与其在调用函数时将寄存器保存到堆栈，您可以切换到其它寄存器。这会加快中断处理。
   - [/E]
     ：在进入中断后马上允许多重中断。这将提高中断的反应速度。
   - [/V]
     ：生成固定向量的向量表。
3. 可在声明时指定中断向量编号。

规则：

1. 若在进行编译时声明了具有参数的中断处理函数，则会输出警告。
2. 若在进行编译时声明了会返回值的中断处理函数，则会输出警告。确保将函数的任何返回值声明为 void 类型。
3. 只有在 #pragma INTERRUPT 之后定义的函数有效。
4. 若指定的不是函数名称，则不会进行任何处理。
5. 重复声明 #pragma INTERRUPT 不会产生错误。
6. 可以同时指定切换 /E 和切换 /B。
7. 若在相同的中断处理函数中编写了不同的中断向量编号，则最后声明的向量编号有效。

```
#pragma  INTTERUPT  intr(vect=10)
#pragma  INTTERUPT  intr(vect=20)                    /* 中断向量编号 20 有效。*/
```

附图 B.77　　编写不同的中断向量编号的范例

#pragma INTERRUPT

声明中断函数

规则：

8. 在 #pragma INTERRUPT 中使用下列任何一项声明指定的函数，将发生编译警告：
   - #pragma ALMHANDLER
   - #pragma INTHANDLER
   - #pragma HANDLER
   - #pragma CYCHANDLER
   - #pragma TASK

范例：

```
extern int      int_counter;

#pragma         INTERRUPT  /B  i_func

void            i_func(void)
{
                int_counter += 1;
}
```

附图 B.78　#pragma INTERRUPT 声明的范例

注意：

1. 若要使用产品随附的启动文件，必须在使用前修改向量段的内容。有关如何修改的详情，请参考 “第 2 章准备启动程序”。

#pragma PARAMETER

声明通过寄存器传递参数的汇编器函数

功能： 声明通过寄存器传递参数的汇编器函数

语法： #pragma PARAMETER △ [/C] △汇编器函数名称 （寄存器名称 , 寄存器名称 , ...）

描述： 这个函数声明当调用汇编器函数时，通过寄存器来传递其参数。

- float 类型， long 类型 （32 位寄存器）：R2R0 和 R3R1
- far 指针类型 （24 位寄存器）：R2R0、 R3R1、 A1 和 A0
- near 指针类型 （16 位寄存器）：A0、 A1、 R0、 R1、 R2 和 R3、 SB
- char 类型和 _Bool 类型 （8 位寄存器）：R0L、 R0H、 R1L 和 R1H
- 寄存器名称无大小写区分。
- 不可声明 long long 类型（64 位整数类型）和 double 类型，以及 structure 和 union 类型。

此外，下列切换可在进行声明时指定。

- [/C]

  通过指定切换 [/c]，将能在调用函数时，并在进入该函数前，生成代码以将寄存器堆栈。（仅限于 NC308WA）

规则：

1. 总是将汇编器函数的原型声明放在 #pragma PARAMETER 的声明之前。若未进行原型声明，则将输出警告，并忽略 #pragma PARAMETER。
2. 在原型声明中须遵守下列规则：
   a. 也请注意在原型声明中指定的参数数量必须与 #pragma PARAMETER 声明中的数量一致。
   b. 下列类型无法在 #pragma PARAMETER 声明中声明为汇编器函数的参数：
      - structure 类型和 union 类型
      - double 类型 long long 类型
   c. 下面显示无法声明的汇编器函数：
      - 返回 structure 或 union 类型的函数
3. 对于 #pragma PARAMETER 所指定函数的输出汇编器名称，将总是在其前面加上 _（下划线）。

范例：

```
int     asm_func(unsigned int, unsigned int);      ← 汇编器函数的
#pragma PARAMETER  asm_func(R0, R1)                原型声明

void    main(void)
{
        int         i, j;

        i = 0x7FFD;
        j = 0x007F;

        asm_func( i, j );                          ← 调用汇编器函数
```

附图 B.79　#pragma PARAMETER 声明的范例

#pragma SPECIAL

声明 special 页子例程调用函数

功能： 声明 special 页子例程调用 （JSRS 指令）函数

语法：
1. #pragma SPECIAL △编号△函数名称 ()
2. #pragma SPECIAL △函数名称 （vect = 编号）

描述：
1. 使用 #pragma SPECIAL 声明的函数，将映射到通过将 special 页向量表中所设置的地址加上 0F0000H 而得到的地址，因此会取决于 special 页子例程调用。
2. 可以在这项声明中指定调用编号：
3. 可以自动生成 Special 页的向量表。

规则：
1. 使用 #pragma SPECIAL 声明的函数将会映射至 program_S 段。确保在 0F0000H 和 0FFFFFH 之间映射 program_S 段。
2. 只能以十进制在 18 和 255 之间的编号调用。
3. 作为标签，“_SPECIAL_calling-number:”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 页子例程表中设置此标签。[6]
   当指定了 -fmake_special_table (-fMST) 选项时，就不需要进行上述设置。
4. 若在函数中编写了不同的调用编号，则最后编写的调用编号有效。

```
#pragma   SPECIAL  func(vect=20)
#pragma   SPECIAL  func(vect=30)        // 调用编号 30 有效
```

附图 B.80　编写不同调用编号的范例

5. 如果要在某个文件中定义函数，而在另一个文件中定义函数的调用，则确保同时在两个文件中编写此声明。

范例：

```
#pragma   SPECIAL  20  func()
void      func(unsigned int, unsigned int);

void      main(void)
{
          int i, j;

          i = 0x7FFD;
          j = 0x007F;

          func( i, j );                 ← special 页子例程调用
}
```

附图 B.81　#pragma SPECIAL 的声明范例

6. 若使用的是随附的启动文件，则修改 fvector 段的内容。有关如何修改启动文件的详情，请参阅 NC30 用户手册操作部分的第 2.2 章 “修改启动程序”。

附录 B.7.4　　使用 MR30 扩展功能

NC30 具有下列可支持实时操作系统 MR30 的扩展功能。

#pragma ALMHANDLER

警报处理程序声明

功能： 声明用于 M16C 系列警报处理程序的实时操作系统

语法： #pragma ALMHANDLER △警报处理程序名称

描述： 通过使用上述格式来声明以 C 编写的警报处理程序（函数），NC30 将为警报处理程序生成代码，以在进入和退出函数时使用。

- 警报处理程序由 JSR 指令从系统时钟中断调用，然后由 RTS 或 EXITD 指令返回。

规则：

1. 不可编写具有参数的警报处理程序。
2. 警报处理程序的返回值必须声明为 void 类型。
3. 函数定义只有在 #pragma ALMHANDLER 之后出现才有效。
4. 若指定的不是函数名称，则不会进行任何处理。
5. 重复声明 #pragma ALMHANDLER 不会产生错误。
6. 在 #pragma ALMHANDLER 中使用下列任何一项声明指定的函数，将发生编译警告：
   - #pragma INTERRUPT
   - #pragma INTHANDLER
   - #pragma HANDLER
   - #pragma CYCHANDLER
   - #pragma TASK

范例：

```
#include  <mrXXX.h>
#include  "id.h"
#pragma  ALMHANDLER  alm

void      alm(void)              ← 确保声明为 void 类型。
{
            :
          （已省略）
            :
}
```

附图 B.82　　#pragma ALMHANDLER 声明的范例

#pragma CYCHANDLER

循环处理程序的声明

功能： 声明用于 M16C 系列周期处理程序的实时操作系统

语法： #pragma CYCHANDLER △周期处理程序名称

描述： 通过使用上述格式来声明以 C 编写的周期处理程序（函数），NC30 将为周期处理程序生成代码，以在进入和退出函数时使用。

- 周期处理程序由 JSR 指令从系统时钟中断调用，然后由 RTS 或 EXITD 指令返回。

规则：

1. 不可编写具有参数的周期处理程序。
2. 周期处理程序的返回值必须声明为 void 类型。
3. 函数定义只有在 #pragma CYCHANDLER 之后出现才有效。
4. 若指定的不是函数名称，则不会进行任何处理。
5. 重复声明 #pragma CYCHANDLER 不会产生错误。
6. 在 #pragma CYCHANDLER 中使用下列任何一项声明指定的函数，将发生编译警告：
   - #pragma INTERRUPT
   - #pragma INTHANDLER
   - #pragma HANDLER
   - #pragma ALMHANDLER
   - #pragma TASK

范例：

```
#include      <mrXXX.h>
#include      "id.h"
#pragma       CYCHANDLER  cyc

void          cyc(void)                ← 确保声明为 void 类型。
{
                  :
              （已省略）
                  :
}
```

附图 B.83　#pragma CYCHANDLER 声明的范例

#pragma INTHANDLER(#pragma HANDLER)

中断处理程序声明

功能： 声明用于 M16C 系列 OS 相关的中断处理程序的实时操作系统

语法：

1. #pragma INTHANDLER △ [/E] △中断处理程序名称
2. #pragma HANDLER △ [/E] △中断处理程序名称

描述：

1. 通过使用上述格式来声明以 C 编写的中断处理程序 （函数）， NC30 将为下面所示的处理生成代码，以在进入和退出函数时使用：
   - 在进入函数时：
     将寄存器推入 （即保存）当前堆栈。
   - 在退出函数时：
     使用 ret_int 系统调用从中断返回。另外当在函数中的返回语句返回时，同样也将通过 ret_int 系统调用从中断返回。
2. 可在声明时指定下列切换。
   - [/E]
     当主控权切换到此函数所声明的中断处理程序后，多重中断即被允许。
3. 若要声明 MR30 OS 相关的中断处理程序，则使用 #pragma INTERRUPT。

规则：

1. 不可编写具有参数的中断处理程序。
2. 中断处理程序的返回值必须声明为 void 类型。
3. 请勿从 C 使用 ret_int 系统调用。
4. 函数定义只有在 #pragma INTHANDLER 之后出现才有效。
5. 若指定的不是函数名称，则不会进行任何处理。
6. 重复声明 #pragma INTHANDLER 不会产生错误。
7. 在 #pragma INTHANDLER 中使用下列任何一项声明指定的函数，将发生编译警告：
   - #pragma INTERRUPT
   - #pragma HANDLER
   - #pragma ALMHANDLER
   - #pragma CYCHANDLER
   - #pragma TASK

范例：

```
#include      <mrXXX.h>
#include       "id.h"
#pragma       INTHANDLER  hand

void          hand(void)
{
                    :
              （已省略）
                    :
              /* ret_int(); */
}
```

附图 B.84　#pragma INTHANDLER 声明的范例

#pragma TASK

任务开始函数声明

功能： 声明用于 M16C 系列任务开始函数的实时操作系统

语法： #pragma TASK △任务开始函数名称

描述： 通过使用上述格式来声明以 C 编写的任务开始函数， NC30 将为处理下面所示任务生成代码，以在进入和退出函数时使用。

- 在退出函数时：
  由 ext_tsk 系统调用结束。另外即使当在函数中的返回语句返回时，同样也将使用 ext_tsk 系统调用返回。

规则：

1. 不需要使用 ext_tsk 系统调用来从任务返回。
2. 任务的返回值必须声明为 void 类型。
3. 函数定义只有在 #pragma TASK 之后出现才有效。
4. 若指定的不是函数名称，则不会进行任何处理。
5. 重复声明 #pragma TASK 不会产生错误。
6. 在 #pragma TASK 中使用下列任何一项声明指定的函数，将发生编译警告：
   - #pragma INTERRUPT
   - #pragma INTHANDLER
   - #pragma HANDLER
   - #pragma ALMHANDLER
   - #pragma CYCHANDLER

范例：

```
#include      <mrXXX.h>
#include      "id.h"
#pragma       TASK  main
#pragma       TASK  tsk1

void          main(void)              ← 确保声明为 void 类型。
{
                  :

              （已省略）
                  :
              sta_tsk(ID_idle);
              sta_tsk(ID_tsk1);
              /* ext_tsk(); */        ← 不须使用 ext_tsk。
}

void          tsk1(void)
                  :
              （其余省略）
```

附图 B.85　#pragma TASK 声明的范例

附录 B.7.5 其它扩展功能

NC30 包含下列用于直接插入嵌入汇编器描述的扩展功能。

#pragma __ASMMACRO

汇编器宏函数

功能： 声明由汇编器宏定义的函数。

语法： #pragma __ASMMACRO △函数名称 (寄存器名称 , ...)

规则：

1. 总是在 #pragma __ASMMACRO 声明之前进行原型声明。确保将汇编器宏函数声明为“static”。
2. 无法声明没有参数的函数。由于参数是通过寄存器传递的，因此请指定适合参数类型的寄存器。
3. 请在汇编器宏名称的定义前面加上下划线 （“_”）。
4. 下面是有关返回值的调用规则。您无法将返回值声明为 structure 和 union 类型。 char 和 _Bool 类型； R0L float 类型； R2R0 int 和 short 类型； R0 double 类型； R3R2R1R0 long 类型； R2R0 long-long 类型； R3R1R2R0。
5. 若更改了寄存器的数据，则须在汇编器宏函数的进入处理将寄存器保存到堆栈，然后在退出处理恢复所保存的寄存器。

范例：

```
static long     mul( int, int );                /* 确保声明 “static” */

#pragma         __ASMMACRO mul(R0,R2)
#pragma         ASM
                _mul    .macro
                mul.w  R2,R0                    ; 返回值被设置到 R2R0 寄存器
.endm
#pragma         ENDASM

long            l;

void            test_func( void )
{
                l = mul( 2, 3 );
}
```

附图 B.86 #pragma __AMMACRO 的范例

#pragma ASM, #pragma ENDASM

直接插入汇编

功能： 在 C 中指定汇编代码。

语法：
#pragma ASM
汇编语句
#pragma ENDASM

描述： 这将输出 #pragma ASM 和 #pragma ENDASM 之间的行，而不修改所生成的汇编源文件。在编写 #pragma ASM 时，确保总是和 #pragma ENDASM 一同使用。否则本编译器会找不到与 #pragma ASM 对应的 #pragma ENDASM，而暂停处理。

规则：
1. 在汇编语言的描述中，请勿编写将导致寄存器内容受损的语句。当编写这类语句时，确保使用 push 和 pop 指令来保存及恢复寄存器的内容。
2. 在 “#pragma ASM” 到 “#pragma ENDASM” 的段内，请勿引用参数和 auto 变量。
3. 在 “#pragma ASM” 到 “#pragma ENDASM” 的段内，请勿编写转移语句 （包括条件转移），因为会影响程序流程。

范例：

```
void        func(void)
{
            int         i, j;

            for(i=0; i < 10;i++){
                        func2();
}

#pragma     ASM
            FCLR        I
LOOP1:
            MOV.W       #0FFH,R0
              :
            （已省略）
              :
            FSET  I

#pragma     ENDASM
}
```

这个部分将直接输出到汇编语言文件。

附图 B.87　#pragma ASM(ENDASM) 的范例

补充： C 预处理器所处理的正是在 #pragma ASM 和 #pragma ENDASM 之间编写的这个汇编语言程序。

#pragma JSRA

使用 JSR.A 来调用函数

功能： 使用 JSR.A 指令来调用函数。

语法： #pragma JSRA △函数名称

描述： 通过 JSR.A 指令调用所有使用 #pragma JSRA 进行声明的函数。当函数包含使用 -fJSRW 选项来进行声明，且在连接时发生错误的代码时，可以指定 #pragma JSRA 以避免发生错误。

规则： 当未指定 -fJSRW 选项时，这项预处理指令不起作用。

范例：

```
extern void     func(int i);
#pragma         JSRA  func()

void            main(void)
{
                func(1);
}
```

附图 B.88　#pragma JSRA 的范例

#pragma JSRW

使用 JSR.W 来调用函数

功能： 使用 JSR.W 指令来调用函数。

语法： #pragma JSRW △函数名称

规则： 默认情况下，当调用相同文件中没有主体定义的函数时，将会使用 JSR.A 指令。不过，使用 #pragma JSRW 进行声明的函数则一直都是通过 JSR.W 来调用。这个指令可帮助缩减 ROM 的大小。

规则：
1. 不可以为 static 函数指定 #pragma JSRW。
2. 当使用 JSR.W 指令的函数调用未达到通过 #pragma JSRW 进行声明的函数时，则将会在连接时发生错误。在这种情况下，不可使用 #pragma JSRW。

范例：

```
extern void func(int i);
#pragma         JSRW  func()

void            main(void)
{
                func(1);
}
```

附图 B.89 #pragma JSRW 的范例

注意： #pragma JSRW 只在直接调用函数时有效。它在进行间接调用时则无效。

#pragma PAGE

输出 .PAGE

功能： 声明新的页在汇编器生成的列表文件中的位置。

语法： #pragma PAGE

描述： 在 C 源代码中使用 #pragma PAGE 行，将会使编译器生成的汇编代码的对应行输出 .PAGE 伪指令。这个指令将生成 PAGE 的指定更动到汇编目录文件。

规则：
1. 不能指定在汇编器伪指令 .PAGE 的标题中所指定的字符串。
2. 不能在 auto 变量声明中编写 #pragma PAGE。

范例：

```
void    func(void)
{
        int     i, j;

        for(i=0; i < 10;i++){
                func2();
        }
#pragma PAGE
        i++;
}
```

附图 B.90　#pragma PAGE 的范例

附录 B.8　　汇编器宏函数

附录 B.8.1　　汇编器宏函数概述

NC30 允许将部分的汇编器命令编写为 C 语言函数。由于特定的汇编器命令能够直接在 C 语言程序中编写，因此可以轻松的调整程序。

附录 B.8.2　　汇编器宏函数的描述范例

如下所示，汇编器宏函数可在 C 语言程序中，使用和 C 语言函数相同的格式来编写。

```
#include  <asmmacro.h>                    /* 包含汇编器宏函数定义文件 */
long              l;
char              a[20];
char              b[20];

void              func(void)
{
                  l = rmpa_b(0,19,a,b);          /* asm 宏函数 （rmpa 命令）*/
}
```

附图 B.91　　汇编器宏函数的描述范例

附录 B.8.3 可通过汇编器宏函数进行编写的命令

下面显示可使用汇编器宏函数来进行编写的汇编器命令，以及它们作为汇编器宏函数的功能与格式。

ABS

功能： 绝对

语法：

```
#include <asmmacro.h>

static signed char abs_b( signed char val );    /* 当以 8 位计算时 */
static signed int abs_w( signed int val );    /* 当以 16 位计算时 */
```

DADC

功能： 返回带进位的 val1 加 val2 的十进制加法的结果。

语法：

```
#include <asmmacro.h>

static char dadc_b( char val1, char val2 );    /* 当以 8 位计算时 */
static int dadc_w( unsigned int val1, unsigned int val2 ); /* 当以 16 位计算时 */
```

DADD

功能： 返回无进位的 val1 加 val2 的十进制加法的结果。

语法：

```
#include <asmmacro.h>

static char dadd_b(char val1, char val2);    /* 当以 8 位计算时 */
static int dadc_w(int val1, int val2);      /* 当以 16 位计算时 */
```

DIV

功能： 为被除数 val2 除以除数 val1 的除法返回余数，包括符号。

语法：
```
#include <asmmacro.h>

static signed char div_b(signed int val1, signed int val2);
/* 以 8 位计算，带符号 */

/ static signed int div_w(signed int val1, signed long val2);
/* 以 16 位计算，带符号 */
```

DIVU

功能： 为被除数 val2 除以除数 val1 的除法返回余数，不包括符号。

语法：
```
#include <asmmacro.h>

unsigned char divu_b(unsigned char val1, unsigned int val2);
/* 以 8 位计算，无符号 */

unsigned int divu_w(unsigned int val1, unsigned long val2);
/* 以 16 位计算，无符号 */
```

DIVX

功能： 为被除数 val2 除以除数 val1 的除法返回余数，不包括符号。

语法：
```
#include <asmmacro.h>

static unsigned char divx_b(unsigned char val1, unsigned int val2 );
/* 以 8 位计算，无符号 */

static unsigned int divx_w( unsigned int val1, unsigned long val2 );
/* 以 16 位计算，无符号 */
```

MOD, MODU

功能： 将 val1 除以 val2 并取得 mod。

语法： #include <asmmacro.h>

```
signed char mod_b(int val1,char val2);  /* 当以 8 位计算时 */
signed int mod_w(long val1,int val2);  /* 当以 16 位计算时 */
unsigned char modu_b(int val1,char val2);  /* 当以无符号的 8 位计算时 */
unsigned int modu_w(unsigned long val1,unsigned int val2);
/* 当以无符号的 16 位计算时 */
```

DSBB

功能： 返回带借位的 val2 减 val1 的十进制减法的结果。

语法： #include <asmmacro.h>

```
static char dsbb_b(char val1, char val2);    /* 当以 8 位计算时 */

static int dsbb_w(int val1, int val2);    /* 当以 16 位计算时 */
```

DSUB

功能： 返回无借位的 val2 减 val1 的十进制减法的结果。

语法： #include <asmmacro.h>

```
static char dsub_b(char val1, char val2);    /* 当以 8 位计算时 */
static int dsub_w(int val1, int val2);    /* 当以 16 位计算时 */
```

MOVdir

功能： 从 val1 转移半字节到 val2。

语法： #include <asmmacro.h>

```
static unsigned char movll(unsigned char val1,unsigned char val2);
/* 从 val1 的高位到 val2 的低位 */
static unsigned char movlh(unsigned char val1,unsigned char val2);
/* 从 val1 的低位到 val2 的高位 */
static unsigned char movhl(unsigned char val1, unsigned char val2);
/* 从 val1 的高位到 val2 的低位 */
static unsigned char movhh(unsigned char val1,unsigned char val2);
/* 从 val1 的高位到 val2 的高位 */
```

RMPA

功能： 初始值：init ；次数：count。假设 p1 和 P2 为存储乘数的起始地址，返回积的总和的运算结果。

语法： #include <asmmacro.h>

```
static long rmpa_b(singed int init, int count, char *p1, char *p2);
/* 当以 8 位计算时 */
static long rmpa_w(long init, int count, int *p1, int *p2);
/* 当以 16 位计算时 */
```

SMOVF

功能： 字符串将按照地址递增指令中的 count 所表示的次数，从 p1 的源地址转移到 p2 的目标地址。无返回值。

语法： #include <asmmacro.h>

```
void smovf_b(char *p1,char *p2,unsigned int count);  /* 以 8 位计算 */
void smovf_w(int *p1,int *p2,unsigned int count);  /* 以 16 位计算 */
```

SHA

功能： val 的值会在根据 count 的次数进行算术移位后返回 。

语法： #include <asmmacro.h>

```
/ static unsigned char sha_b(signed char count, unsigned char val);
/* 当以 8 位计算时 */
static unsigned int sha_w(signed char count, unsigned int val);
/* 当以 16 位计算时 */
static unsigned long sha_l(signed char count, unsigned long val);
/* 当以 32 位计算时 */
```

SHL

功能： val 的值会在根据 count 的次数进行逻辑移位后返回 。

语法： #include <asmmacro.h>

```
static unsigned char shl_b(signed char count, unsigned char val);
/* 当以 8 位计算时 */
static unsigned int shl_w(signed char count, unsigned int val);
/* 当以 16 位计算时 */
static unsigned long shl_l(signed char count, unsigned long val);
/* 当以 24 位计算时 */
```

SMOVB

功能： 字符串将按照地址递减指令中的 count 所表示的次数，从 p1 的源地址转移到 p2 的目标地址。这项指定没有返回值。

语法： #include <asmmacro.h>

```
static void smovb_b(char _far *p1, char _far *p2, unsigned int count);
/* 以 8 位计算时 */
static void smovb_w(int _far *p1, int _far *p2, unsigned int count);
/* 当以 16 位计算时 */
```

SSTR

功能： 以 val 作为所要存储的数据、p 作为 val 地址的转移目标地址及 count 作为数据的转移次数以存储字符串。这项指定没有返回值。

语法：

```
#include <asmmacro.h>

static void sstr_b(char val, char _far *p, unsigned int count);
/* 以 8 位计算时 */
static void sstr_w(int val, int _far *p, unsigned int count);
/* 以 16 位计算时 */
```

ROLC

功能： 返回左移 1 位的 val 值，包括 C 标志。

语法：

```
#include <asmmacro.h>

static unsigned char rolc_b(unsigned char val1);
/* 当以 8 位计算时 */
static unsigned int rolc_w(unsigned int val1);
/* 当以 16 位计算时 */
```

RORC

功能： 返回右移 1 位的 val 值，包括 C 标志。

语法：

```
#include <asmmacro.h>

static unsigned char rorc_b(unsigned char val);
/* 当以 8 位计算时 */
static unsigned int rorc_w(unsigned int val);
/* 当以 16 位计算时 */
```

ROT

功能： val 的值会在根据 count 的次数进行移动后返回 。

语法： #include <asmmacro.h>

```
static unsigned char rot_b(signed char count, unsigned char val);
/* 当以 8 位计算时 */
static unsigned int rot_w(signed char count, unsigned int val);
/* 当以 16 位计算时 */
```

NEG

功能： 求反。

语法： #include <asmmacro.h>

```
signed char neg_b(signed char val);
/* 当以 8 位计算时 */
signed int neg_w(signed int val);
/* 当以 16 位计算时 */
```

NOT

功能： 非。

语法： #include <asmmacro.h>

```
signed char not_b(signed char val);
/* 当以 8 位计算时 */
signed int not_w(signed int val);
/* 当以 16 位计算时 */
```

附录 C　C 语言规格说明的概述

除了推出标准版本的 C 外， C 语言的规格说明也包含用于嵌入式系统的扩展函数。

附录 C.1　性能规格说明

附录 C.1.1　标准规格说明概述

本编译器是一个主要用于 M16C/60、 M16C/30、 M16C/20、 M16C/10、 R8C/Tiny 系列的交叉 C 编译器。在语言的规格说明方面，它基本上与标准的全套 C 语言相同，但也同时具有用于 M16C/60、 M16C/30、M16C/20、 M16C/10、 R8C/Tiny 系列硬件的规格说明和用于嵌入式系统的扩展函数。

- 用于嵌入式系统的扩展函数 （near/far 修饰符，及 asm 函数等）
- 标准程序库中包含了浮点程序库和主机相关的函数。

附录 C.1.2　NC30 性能的简介

本节提供有关 NC30 性能的概述。

(a)　测试环境

附表 C.1 中显示标准的 PC 环境。

附表 C.1　标准的 PC 环境

项目	PC 类型	OS 版本
PC 环境	IBM 或兼容型 PC/AT	Windows XP
CPU 类型	Pentium IV	
存储器	至少 128MB （使用 HEW）	

(b)　编写 C 源文件的规格说明

附表 C.2 中显示编写 NC30 C 源文件的规格说明。对于无法实际测量的项目，将提供估计值。

附表 C.2　编写 C 源文件的规格说明

项目	规格说明
源文件中每行的字符数	512 字节 （字符）包括新行代码
源文件中的行数	最多 65535

(c) NC30 的规格说明

附表 C.3 到附表 C.4 列出了 NC30 的规格说明。对于无法实际测量的项目，将提供估计值。

附表 C.3 NC30 的规格说明 (1)

项目	规格说明
可在 NC30 中指定的最大文件数量	无限定 （视存储器容量而定）
最大文件名长度	视操作系统而定
可在 nc30 命令行选项 -D 中指定的最大宏数量	无限定 （视存储器容量而定）
可在 nc30 命令行选项 -I 中指定的最大目录数量	最多 50
可在 nc30 命令行选项 -as30 中指定的最大参数数量	无限定 （视存储器容量而定）
可在 nc30 命令行选项 -ln30 中指定的最大参数数量	无限定 （视存储器容量而定）
复合语句、迭代控制结构和选择控制结构的最大嵌套级	无限定 （视存储器容量而定）
条件编译的最大嵌套级	无限定 （视存储器容量而定）
修改所声明的基本类型、数组和函数声明符的指针数量	无限定 （视存储器容量而定）
函数定义的数量	无限定 （视存储器容量而定）
在一个块范围内的标识符数量	无限定 （视存储器容量而定）
可在一个源文件中同时指定的最大宏标识符数量	无限定 （视存储器容量而定）
宏名称替换的最大数量	无限定 （视存储器容量而定）
输入程序中的逻辑源的行数量	无限定 （视存储器容量而定）
#include 文件的最大嵌套级	最多 40
一个 switch 语句中的最大 case 名称数量 （在没有 switch 语句嵌套的情况下）	无限定 （视存储器容量而定）
可在 #if 和 #elif 中定义的运算符和操作数总数	无限定 （视存储器容量而定）
可为每个函数保留的堆栈帧大小 （按字节计算）	最多 64K 字节
可在 #pragma ADDRESS 中定义的变量数量	无限定 （视存储器容量而定）
括号的最大嵌套级	无限定 （视存储器容量而定）
在初始化表达式中定义变量时可定义的初始值数量	无限定 （视存储器容量而定）
修饰声明符的最大嵌套级	视 YACC 的堆栈大小而定
声明符括号的最大嵌套级	视 YACC 的堆栈大小而定
运算符括号的最大嵌套级	视 YACC 的堆栈大小而定
每个内部标识符或宏名称的最大有效字符数	无限定 （视存储器容量而定）
每个外部标识符的最大有效字符数	无限定 （视存储器容量而定）
每个源文件的最大外部标识符数量	无限定 （视存储器容量而定）

附表 C.4　　NC30 的规格说明 (2)

项目	规格说明
在每个块范围内的最大标识符数量	无限定 （视存储器容量而定）
每个源文件的最大宏数量	无限定 （视存储器容量而定）
每个函数调用和每个函数的最大参数数量	无限定 （视存储器容量而定）
每个宏的最大参数或宏调用参数数量	最多 31
字符串文字在串联后的最大字符数	无限定 （视存储器容量而定）
目标大小的最大值 （按字节计算）	无限定 （视存储器容量而定）
每个结构 / 联合的最大成员数量	无限定 （视存储器容量而定）
每个枚举符的最多枚举符常数数量	无限定 （视存储器容量而定）
每个 struct 声明列表的最大结构或联合的嵌套级	无限定 （视存储器容量而定）
每个字符串的最大字符数	视操作系统而定
每个文件的最大行数	无限定 （视存储器容量而定）

附录 C.2 标准语言的规格说明

本章将论述有关使用标准语言规格说明的 NC30 语言规格说明。

附录 C.2.1 语法

本节将描述语法标志元素。在 NC30 中，下列项目将作为标志处理：

- 关键字
- 常数
- 运算符
- 注释
- 标识符
- 字符文字
- 标点

(a) 关键字

NC30 将下列项目解释为关键字。

附表 C.5 关键字列表

_asm	_far	_near	asm	auto
_Bool	break	case	char	const
continue	default	do	double	else
enum	extern	far	float	For
goto	if	inline	int	long
near	register	restrict	return	short
signed	sizeof	static	struct	switch
union	unsigned	void	volatile	while
typedef	-	-	-	-

(b) 标识符

标识符包含下列元素：

- 第一个字符是字母或下划线 （A 到 Z、 a 到 z，或 __）
- 第二个和接下来的字符是字母数字或下划线 （A 到 Z、 a 到 z、 0 到 9，或 __）

标识符可包含多达 200 个字符。不过，不可在标识符中指定日文字符。

(c) 常数

常数包含下列项目：

- 整数常数
- 浮点常数
- 字符常数

(1) 整数常数

除了十进制外，您也可以指定八进制和十六进制的整数常数。附表 C.6 中显示每一种基数 （十进制、八进制和十六进制）的格式。

附表 C.6 指定整数常数

基数	表示法	结构	范例
十进制	无	0123456789	15
八进制	从 0 （零）开始	01234567	017
十六进制	以 0X 或 0x 开头	0123456789ABCDEF 0123456789abcdef	0XF 或 0xf
二进制数字	以 0b 或 0B 开头	01	0b1 或 0B1

按照下列顺序，根据值来确定整数常数的类型。

- 八进制、十六进制和二进制的数字：
  signed int → unsigned int → signed long → unsigned long → signed long long → unsigned long long
- 十进制：
  signed int → signed long → signed long long

添加 U 或 u、L 或 l、 LL 或 ll 的后缀，以便对整数常数进行如下的处理：

1. 无符号常数
   通过在值之后添加字母 U 或 u 来指定无符号常数。值的类型将根据下列顺序来确定：
   unsigned int → unsigned long → unsigned long long
2. long 类型常数
   通过添加字母 L 或 l 来指定 long 类型常数。值的类型将根据下列顺序来确定：
   - 八进制、十六进制和二进制的数字：
     signed long → unsigned long → signed long long → unsigned long long
   - 十进制：
     signed long long → unsigned long long
3. long long 类型常数
   通过添加字母 LL 或 ll 来指定 long long 类型常数。值的类型将根据下列顺序来确定：
   - 八进制、十六进制和二进制的数字：
     signed long long → unsigned long long
   - 十进制：
     signed long long

(2) 浮点常数

若值未添加任何后缀，则浮点常数将作为 double 类型处理。若要使它们作为 float 类型处理，必须在值之后加上字母 F 或 f。若添加了 L 或 l，它们将作为 long double 类型处理。

(3) 字符常数

字符常数通常以单引号编写，如 ‘character’。您也可以包含下列扩展表示法 （转义符 （Escape Sequence）和三字符组 （Trigraph Sequence））。十六进制的值通过在值之前加上 \x 来表示。八进制的值通过在值之前加上 \ 来表示。

附表 C.7　　扩展表示法列表

表示法	转义符	表示法	三字符组
\'	单引号	\ 常数	八进制
\"	引号	\x 常数	十六进制
\\	反斜线	??(	表示 “[” 字符
\?	问号	??/	表示 “\” 字符
\a	响铃	??)	表示 “]” 字符
\b	退格	??'	表示 “^” 字符
\f	换页	??<	表示 “{” 字符
\n	换行	??!	表示 “{” 字符
\r	回车	??>	表示 “}” 字符
\t	水平制表符	??–	表示 “~” 字符
\v	垂直制表符	??=	表示 “#” 字符

(d)　字符文字

字符文字编写在双引号中，如 “字符串”。附表 C.7 中所示用于字符常数的扩展表示法也可以用于字符文字。

(e)　运算符

NC30 可理解附表 C.8 中所显示的运算符。

附表 C.8　　运算符列表

一元运算符	+ +	逻辑运算符	&&
	– –		¦ ¦
	–		!
二进制运算符	+	条件运算符	?:
	–	逗号运算符	,
	*	地址运算符	&
	/	指针运算符	*
	%	按位运算符	<<
赋值运算符	=		>>
	+=		&
	–=		¦
	*=		^
	/=		–
	%=		&=
关系运算符	>		=
	<		^=
	>=		<<=
	<=		>>=
	= =	sizeof 运算符	sizeof
	!=		

(f) 标点

NC30 将下列项目理解为标点。

- {
- }
- :
- ;
- ,

(g) 注释

在 / * 和 */ 之间的部分是注释。注释无法被嵌套。
在 “//” 和行末之间的部分是注释。

附录 C.2.2 类型

(a) 数据类型

NC30 支持下列数据类型。

- 字符类型
- 整数类型
- structure
- union
- 枚举类型
- void
- 浮点类型

(b) 修饰类型

NC30 将下列项目解释为修饰类型。

- const
- volatile
- restrict
- near
- far

(c) 数据类型和大小

附表 C.9 中显示与数据类型相应的大小。

附表 C.9 数据类型和位大小

类型	是否存在符号	位大小	值范围
_Bool	否	8	0, 1
char unsigned char	否	8	0 至 255
signed char	是	8	-128 至 127
int short signed int signed short	是	16	-32768 至 32767
unsigned int unsigned short	否	16	0 至 65535
long signed long	是	32	-2147483648 至 2147483647
unsigned long	否	32	0 至 4294967295
long long signed long long	是	64	-9223372036854775808 至 9223372036854775807
unsigned long long	否	64	18446744073709551615
float	是	32	1.17549435e-38F 至 3.40282347e+38F
double long double	是	64	2.2250738585072014e-30 至 1.7976931348623157e+30
near pointer	否	16	0 至 0xFFFF
far pointer	否	32	0 至 0xFFFFFFFF

- _Bool 类型不可指定为带符号。
- 若将 char 类型指定为无符号，它将被作为 unsigned char 类型处理。
- 若将 int 或 short 类型指定为无符号，它将被作为 signed int 或 signed short 类型处理。
- 若将 long 类型指定为无符号，它将被作为 signed long 类型处理。
- 若将 long long 类型指定为无符号，它将被作为 signed long long 类型处理。
- 若将结构的位字段成员指定为无符号，它将被作为 unsigned 处理。
- 无法为位字段指定 long long 类型。

附录 C.2.3 表达式

附表 C.10 和附表 C.11 显示了表达式的类型和它们的元素之间的关系。

附表 C.10 表达式的类型和它们的元素 (1)

表达式的类型	表达式的元素
主要表达式	标识符
	常数
	字符文字
	（表达式）
	主要表达式
后置表达式	后置表达式 [表达式]
	后置表达式 （参数列表， …）
	后置表达式 . 标识符
	后置表达式 –> 标识符
	后置表达式 ++
	后置表达式 —
	后置表达式
一元表达式	++ 一元表达式
	— 一元表达式
	一元运算符类型转换表达式
	sizeof 一元表达式
	sizeof （类型名称）
	一元表达式
类型转换表达式	（类型名称）类型转换表达式
	类型转换表达式
表达式	表达式 * 表达式
	表达式 / 表达式
	表达式 % 表达式
加法和减法表达式	表达式 + 表达式
	表达式 – 表达式
按位移位表达式	表达式 << 表达式
	表达式 >> 表达式
关系表达式	表达式
	表达式 < 表达式
	表达式 > 表达式
	表达式 <= 表达式
	表达式 >= 表达式
等价表达式	表达式 = = 表达式
	表达式 != 表达式
按位 “与”	表达式 & 表达式
按位 “异或”	表达式 ^ 表达式
按位 “或”	表达式 | 表达式
逻辑 “与”	表达式 && 表达式
逻辑 “或”	表达式 || 表达式
条件表达式	表达式 ? 表达式 : 表达式

附表 C.11　　表达式的类型和它们的元素 (2)

表达式的类型	表达式的元素
赋值表达式	一元表达式 += 表达式
	一元表达式 –= 表达式
	一元表达式 *= 表达式
	一元表达式 /= 表达式
	一元表达式 %= 表达式
	一元表达式 <<= 表达式
	一元表达式 >>= 表达式
	一元表达式 &= 表达式
	一元表达式 ¦= 表达式
	一元表达式 ^= 表达式
	赋值表达式
逗号运算符	表达式，一元表达式

附录 C.2.4　　声明

声明的类型有两种：

- 变量声明
- 函数声明

(a)　变量声明

使用附图 C.1 中所示的格式来声明变量。

存储类说明符 Δ 类型声明符 Δ 声明说明符 Δ 初始化表达式 ;

附图 C.1　　变量的声明格式

(1)　存储类说明符

NC30 支持下列存储类说明符。

- extern
- static
- typedef
- auto
- register

(2)　类型声明符

NC30 支持下列类型声明符。

- _Bool
- int
- long
- float
- unsigned
- struct
- enum
- char
- short
- long long
- double
- signed
- union

(3) 声明说明符

在 NC30 中使用附图 C.2 中所示的声明说明符格式。

```
Declarator   ：  Pointer opt declarator2
Declarator2  ：  identifier( declarator )
                 declarator2[ 常数表达式 opt ]
                 declarator2( 虚设参数列表 opt )
* 常数表达式中只有显示数组数量的第一个数组可被忽略。
* opt 表示可选项目。
```

附图 C.2 声明说明符的格式

(4) 初始化表达式

NC30 允许在初始化表达式中使用附图 C.3 中所示的初始值。

```
整数类型      ：常数
整数类型数组  ：常数 , 常数 ....
字符类型      ：常数
字符类型数组  ：字符文字 , 常数 ....
指针类型      ：字符文字
指针数组      ：字符文字 , 字符文字 ....
```

附图 C.3 可在初始化表达式中指定的初始值

(b) 函数声明

使用附图 C.4 中所示的格式来声明函数。

- 函数声明 (定义)
  存储类说明符 Δ 类型声明符 Δ 声明说明符 Δ 主程序
- 函数声明 (原型声明)
  存储类说明符 Δ 类型声明符 Δ 声明说明符 ;

附图 C.4 函数的声明格式

(1) 存储类说明符

NC30 支持下列存储类说明符。

- extern
- static

(2) 类型声明符

NC30 支持下列类型声明符。

- _Bool
- int
- long
- float
- unsigned
- struct
- enum
- char
- short
- long long
- double
- signed
- union

(3) 声明说明符

在 NC30 中使用附图 C.5 中所示的声明说明符格式。

Declarator ： Pointer $_{opt}$ declarator2
Declarator2 ： identifier(虚设参数列表 $_{opt}$)
(declarator)
declarator[常数表达式 $_{opt}$]
declarator(虚设函数列表 $_{opt}$)

* 常数表达式中只有显示数组数量的第一个数组可被忽略。
* opt 表示可选项目。
* 虚设函数列表在原型声明中被一个类型声明符列表所替换。

附图 C.5 声明说明符的格式

(4) 程序主体

使用附图 C.6 中所示的程序主体格式

变量声明符 opt 复合语句的列表

* 原型声明中没有程序的主体，并且以分号结束。
* opt 表示可选项目。

附图 C.6 程序主体的格式

附录 C.2.5 语句

NC30 支持下列各项。

- 标号语句
- 表达式 / 空语句
- 迭代语句
- 汇编语言语句
- 复合语句
- 选择语句
- 跳转语句

(a) 标号语句

使用附图 C.7 中所示的标号语句格式

```
标识符      ：语句
case 常数  ：语句
default      ：语句
```

附图 C.7 标号语句的格式

(b) 复合语句

使用附图 C.8 中所示的复合语句格式

```
{ 声明列表 opt 语句列表 opt opt }
* opt 表示可选项目。
```

附图 C.8 复合语句的格式

(c) 表达式 / 空语句

使用附图 C.9 中所示的表达式和空语句格式

```
表达式;
表达式;
空语句;
;
```

附图 C.9 表达式和空语句的格式

(d) 选择语句

使用附图 C.10 中所示的选择语句格式

```
if( 表达式 ) 语句
if( 表达式 ) 语句 else 语句
switch( 表达式 ) 语句
```

附图 C.10 选择语句的格式

(e) 迭代语句

使用附图 C.11 中所示的迭代语句格式

```
while( 表达式 ) 语句
do 语句 while( 表达式 ) ;
for( 表达式 opt ; 表达式 opt ; 表达式 opt ) 语句;

* opt 表示可选项目。
```

附图 C.11 迭代语句的格式

(f) 跳转语句

使用附图 C.12 中所示的跳转语句格式

```
goto 标识符;
continue ;
break ;
return 表达式 opt ;

* opt 表示可选项目。
```

附图 C.12 跳转语句的格式

(g) 汇编语言语句

使用附图 C.13 中所示的汇编语言格式

```
asm( “文字” ) ;
文字：汇编语言语句
```

附图 C.13 汇编语言语句的格式

附录 C.3 预处理命令

预处理符号以井字符号 （#）开头，并由 cpp30 预处理器进行处理。本章提供预处理命令的规格说明。

附录 C.3.1 可用的预处理命令列表

附表 C.12 中列出了在 NC30 中可用的预处理命令。

附表 C.12 预处理命令列表

命令	功能
#assert	当常数表达式的演算结果，与预估值不符合时输出警告。
#define	定义宏。
#elif	执行条件编译。
#else	执行条件编译。
#endif	执行条件编译。
#error	输出信息到标准输出器件，并终止处理。
#if	执行条件编译。
#ifdef	执行条件编译。
#ifndef	执行条件编译。
#include	包含特定文件。
#line	指定文件的行数。
#pragma	指示对本编译器的扩展函数进行处理。
#undef	取消定义宏。

附录 C.3.2 预处理命令参考

NC30 预处理命令在下面有更详细的描述。它们按照在附表 C.12 中的顺序列出。

#assert

功能： 若常数表达式的结果为零 （0），则发出警告。

格式： #assert 常数表达式

描述： 若常数表达式的结果为零 （0），则发出警告。不过，编译将继续进行。

```
[Warning(cpp30.82):x.c, line xx]assertion warning
```

#define

功能： 定义宏。

格式：
1. #define 标识符 字符串 opt
2. #define 标识符 (标识符列表 opt) 字符串 opt

描述：
1. 将标识符定义为宏。
2. 将标识符定义为宏。在此格式中，请勿在第一个标识符和开括号 ‘(’ 之间插入任何空格或制表符。

- 下列代码中的标识符被空白符所替代。

```
#define SYMBOL
```

- 当使用一个宏来定义函数时您可以插入反斜线使代码跨越两行或多行。
- 下面四个标识符是编译器的保留字。

```
_ _FILE_ _ ..... 源文件名称
_ _LINE_ _ ..... 当前源文件行号
_ _DATE_ _ .....编译日期 （格式：mm dd yyyy）
_ _TIME_ _ .....编译时间 （格式：hh:mm:ss）
```

下面是在 NC30 中预定义的宏。

```
M16C （当使用 “-R8C” 选项时，将定义 _ _R8C_ _。）
NC30
```

- 您可以和标志一起使用标志字符串运算符 ‘#’ 及标志连接运算符 ‘##’，如下所示。

```
#define debug(s,t) printf("x"#s" = %d x"#t" = %d",x ## s,x ## t)
当这个宏的参数 debug (s, t) 被指定为 debug (1, 2) 时它们将被解释如下

#define debug(s,t) printf("x1 = %d x2 = %d", x1,x2)
```

- 宏定义可被如下所示进行嵌套最多 20 个嵌套级。

```
#define  XYZ1  100
#define  XYZ2  XYZ1
                :
          ( 已省略 )
                :
#define  XYZ20  XYZ19
```

#error

功能： 暂停编译并将信息输出到标准输出器件。

格式： #error △字符串

描述：
- 暂停编译。
- 找到字符串后这个命令会将字符串输出到标准的输出器件。

#if - #elif - #else - #endif

功能： 执行条件编译。（检查表达式是 “真”或 “假”。）

格式：
#if 常数表达式
:
#elif 常数表达式
:
#else
:
#endif

描述：
- 若常数的值为 “真”（非 0），命令 #if 和 #elif 将处理之后的程序。
- #elif 和 #if、 #ifdef 或 #ifndef 成对使用。
- #else 和 #if 成对使用。请勿在 #else 和换行之间使用任何标志。不过，您可以插入注释。
- #endif 表示 #if 控制范围的结束。当使用了命令 #if 时，总是确保输入 #endif。
- #if - #elif - #else - #endif 的组合可被嵌套。嵌套的级数并未设置限制 （但取决于可用的内存）。

#ifdef - #elif - #else - #endif

功能： 执行条件编译。检查宏是否已被定义。

格式：

#ifdef △标识符
:
#elif △常数表达式
:
#else
:
#endif

描述：

- 若已定义了标识符 #ifdef 将处理之后的程序。您也可以进行以下描述。

```
#if △ !defined △标识符
#if △ !defined △ ( 标识符 )
```

- #else 和 #ifdef 成对使用。请勿在 #else 和换行之间使用任何标志。不过，您可以插入注释。
- #elif 和 #if、 #ifdef 或 #ifndef 成对使用。
- #endif 表示 #ifdef 控制范围的结束。当使用了命令 #ifdef 时，总是确保输入 #endif。
- #ifdef - #else - #endif 的组合可被嵌套。嵌套的级数并未设置限制 （但取决于可用的内存）。

#ifndef - #elif - #else - #endif

功能： 执行条件编译。检查宏是否已被定义。

格式：

#ifndef △标识符
:
#elif △常数表达式
:
#else
:
#endif

描述：

- 若未定义标识符 #ifndef 将处理之后的程序。您也可以进行以下描述。

```
#if △ !defined △标识符
#if △ !defined △ ( 标识符 )
```

- #else 和 #ifndef 成对使用。请勿在 #else 和换行之间使用任何标志。不过，您可以插入注释。
- #elif 和 #if、 #ifdef 或 #ifndef 成对使用。
- #endif 表示 #ifndef 控制范围的结束。当使用了命令 #ifndef 时，总是确保输入 #endif。
- #ifndef - #else - #endif 的组合可被嵌套。嵌套的级数并未设置限制 （但取决于可用的内存）。
- 您不可在常数表达式中使用 sizeof 运算符、 cast 运算符或变量。

#include

功能： 包含指定的文件。

格式：

1. #include △ < 文件名称 >
2. #include △ “文件名称”
3. #include △ 标识符

描述：

1. 从 nc30 的命令行选项 -I 所指定的目录取得 < 文件名称 >。
   若找不到 < 文件名称 >，则从环境变量 “INC30” 所指定的目录中搜索。
2. 从当前目录取得 “文件名称”。若找不到 “文件名称”，则按顺序从下列目录中搜索。
   1. 由 nc30 的启动选项 -I 所指定的目录。
   2. 由环境变量 “INC30” 指定的目录
3. 若被宏扩展的标识符是 < 文件名称 > 或 “文件名称”，这个命令将根据搜索 [1] 或 [2] 的规则从目录取得该文件。

- 最大的嵌套级数是 40。
- 若所指定的文件不存在，将发生包含错误。

#line

功能： 更改文件的行数。

格式： #line △整数△ “文件名称”

描述：

- 指定文件的行号及文件名称。
- 您可以更改源文件的名称和行号。

#pragma

功能： 指示系统处理 NC30 扩展函数。

格式：

1. #pragma ROM △变量名称
2. #pragma SBDATA △变量名称
3. #pragma SECTION △预定的段名称△修改后的段名称
4. #pragma STRUCT △结构的标志名称△ unpack
5. #pragma STRUCT △结构的标志名称△ arrange
6. #pragma EXT4MPTR △指针名称
7. #pragma ADDRESS △变量名称△绝对地址
8. #pragma BITADDRESS 变量名称位，绝对地址
9. #pragma INTCALL △ [/C] △ INT 编号△汇编器函数名称 (寄存器名称，寄存器名称， ...)
10. #pragma INTCALL △ [/C] △ INT 编号△ C 语言函数名称 ()
11. #pragma INTERRUPT △ [/B|/E] △中断向量编号△中断处理函数名称
12. #pragma PARAMETER △ [/C] △汇编器函数名称 (寄存器名称，寄存器名称，...)
13. #pragma SPECIAL △ [/C] △ special 编号 .. 函数名称
14. #pragma ALMHANDLER △ 警报处理器函数名称
15. #pragma CYCHANDLER △周期处理器函数名称
16. #pragma INTHANDLER △中断处理器函数名称
17. #pragma HANDLER △ 中断处理器函数名称
18. #pragma TASK △任务开始函数名称
19. #pragma ASM
20. #pragma ENDASM
21. #pragma JSRA △函数名称
22. #pragma JARW △函数名称
23. #pragma PAGE
24. #pragma __ASMMACRO △函数名称 (寄存器名称)

#pragma

描述：

1. 安排 rom 段的工具
2. 使用 SB 相对寻址来描述变量的工具
3. 更改段的基本名称的工具
4. 控制结构的数组的工具
5. 控制结构的数组的工具
6. 声明存取 4M 字节 ROM 存储区的指针的工具
7. 指定输入 / 输出变量的绝对地址的工具
8. 指定输入 / 输出变量的绝对位地址的工具
9. 声明使用软件中断的函数的工具
10. 声明使用软件中断的函数的工具
11. 编写中断函数的工具
12. 声明通过寄存器传递的汇编器函数的工具
13. 声明 special 页子例程调用函数的工具
14. 声明警报处理器函数的工具
15. 声明周期处理器函数的工具
16. 声明中断处理器函数的工具
17. 声明中断处理器函数的工具
18. 声明任务开始函数的工具
19. 描述直接插入汇编器的工具
20. 描述直接插入汇编器的工具
21. 声明使用 JSR.A 指令来调用的函数的工具
22. 声明使用 JSR.W 指令来调用的函数的工具
23. 输出 .PAGE 的工具
24. 声明汇编器宏函数的工具

- 您只可以使用 #pragma 来指定上述 24 种处理函数。若您在 #pragma 之后指定上述以外的字符串或标识符，该指定将被忽略。
- 默认情况下，若您所指定的 #pragma 函数不被支持，并不会输出警告。只有在您指定 nc30 命令行选项 - Wunknown_pragma (-WUP) 时，才会输出警告。

#undef

功能： 取消一个被定义为宏的标识符。

格式： #undef 标识符

描述：

- 取消一个被定义为宏的标识符。
- 下面四个标识符是编译器的保留字。由于这些标识符必须总是有效，所以请勿使用 #undef 来取消它们的定义。

_ _FILE_ _	源文件的名称
_ _LINE_ _	当前源文件行号
_ _DATE_ _	编译日期 （格式：mm dd yyyy）
_ _TIME_ _	编译时间 （格式：hh:mm:ss）

附录 C.3.3 预定义的宏

下面是在 NC30 中预定义的宏。

- M16C（当使用 “-R8C”选项时，将定义 _ _R8C_ _。）
- NC30

附录 C.3.4 使用预定义的宏

预定义的宏的其中一个使用范例是使用预处理命令来切换非 NC30 的 C 程序中的机器相关代码。

```
#ifdef  NC30
#pragma  ADDRESS  port0  2H
#pragma  ADDRESS  port1  3H
#else
#pragma  AD  portA = 0x5F
#pragma  AD  portA = 0x60
#endif
```

附图 C.14 预定义的宏的使用范例

附录 D　C 语言规格的说明规则

本附录将描述由 NC30 处理的数据的内部结构及映射，符号在运算中的扩展规则等，以及调用函数和函数返回值的规则。

附录 D.1　数据的内部表达

附录 D.1.1　整数类型

附表 D.1 中显示整数类型的数据所使用的字节数。

附表 D.1　整数类型的数据大小

类型	是否存在符号	位大小	值范围
_Bool	否	8	0, 1
char unsigned char	否	8	0 至 255
signed char	是	8	-128 至 127
int short signed int signed short	是	16	-32768 至 32767
unsigned int unsigned short	否	16	0 至 65535
long signed long	是	32	-2147483648 至 2147483647
unsigned long	否	32	0 至 4294967295
long long signed long long	是	64	-9223372036854775808 至 9223372036854775807
unsigned long long	否	64	18446744073709551615
float	是	32	1.17549435e-38F 至 3.40282347e+38F
double long double	是	64	2.2250738585072014e-30 至 1.7976931348623157e+30
near pointer	否	16	0 至 0xFFFF
far pointer	否	32	0 至 0xFFFFFFFF

- _Bool 类型不可指定为带符号。
- 若将 char 类型指定为无符号，它将被作为 unsigned char 类型处理。
- 若将 int 或 short 类型指定为无符号，它将被作为 signed int 或 signed short 类型处理。
- 若将 long 类型指定为无符号，它将被作为 signed long 类型处理。
- 若将 long long 类型指定为无符号，它将被作为 signed long long 类型处理。
- 若将结构的位字段成员指定为无符号，它将被作为 unsigned 处理。
- 无法为位字段指定 long long 类型。

附录 D.1.2 浮点类型

附表 D.2 中显示浮点类型的数据所使用的字节数。

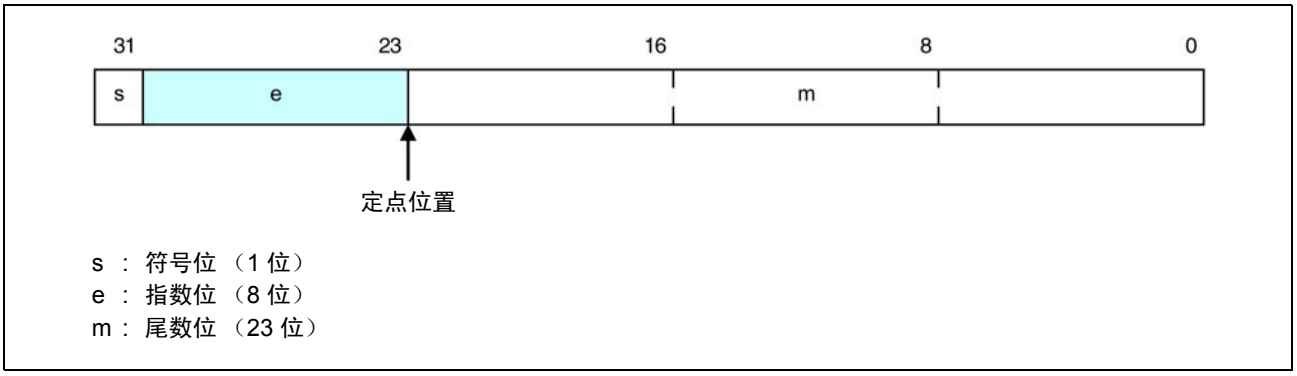
附表 D.2 浮点类型的数据大小

类型	是否存在符号	位大小	值范围
float	是	32	1.17549435e-38F 至 3.40282347e+38F
double long double	是	64	2.2250738585072014e-30 至 1.7976931348623157e+30

NC30 的浮点格式符合 IEEE （美国电气电子工程师协会）的格式标准。下面显示单精度和双精度浮点格式。

(1) 单精度浮点数据格式

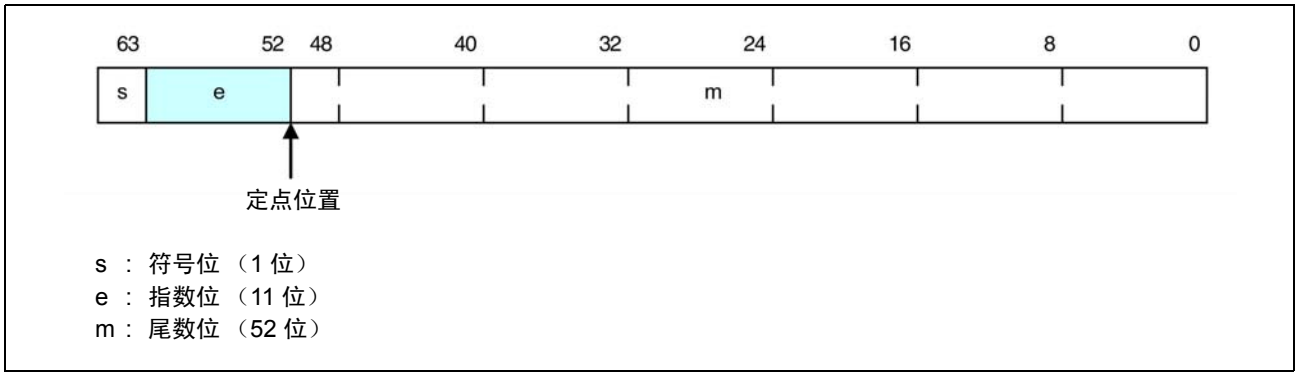
附图 D.1 中显示二进制浮点 （float）数据的格式。



附图 D.1 单精度浮点数据格式

(2) 双精度浮点数据格式

附图 D.2 中显示二进制浮点 （double 和 long double）数据的格式。



附图 D.2 双精度浮点数据格式

附录 D.1.3 枚举类型

枚举类型具有与 unsigned int 类型相同的内部表达。除非另外说明，否则将按成员出现的顺序来应用整数 0、 1、 2.....。

您也可以使用 nc30 命令行选项 -fchar_enumerator (-fCE) 来强制枚举类型具有与 unsigned char 类型相同的内部表达。

附录 D.1.4 指针类型

附表 D.3 中显示指针类型的数据所使用的字节数。

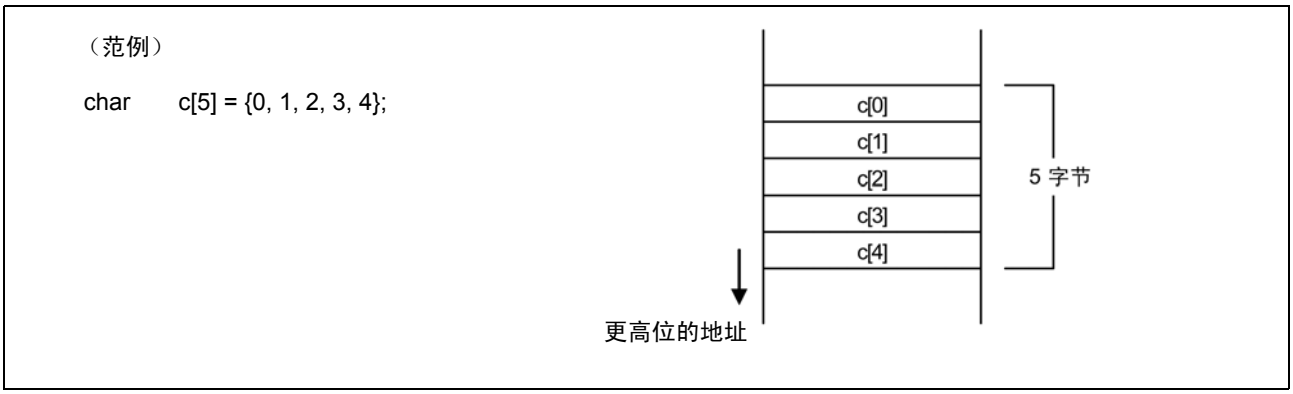
附表 D.3 指针类型的数据大小

类型	是否存在符号	位大小	范围
near 指针	无	16	0 至 0xFFFF
far 指针	无	32	0 至 0xFFFFF

far 指针的 32 位中只有最低有效的 24 位有效。

附录 D.1.5 数组类型

数组类型被连续映射到一个存储区。此区域大小等于元素大小 （按字节计算）与元素数量的乘积。它们按元素出现的顺序映射到存储器。附图 D.3 中是一个映射范例。



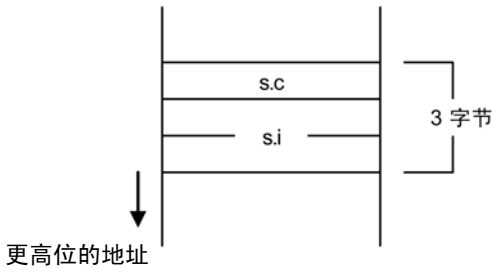
附图 D.3 数组的布局范例

附录 D.1.6 结构类型

结构类型按照其成员数据的顺序连续映射。附图 D.4 中是一个映射范例。

（范例）

```
struct TAG{
        char    c;
        int     i;
        s;
```

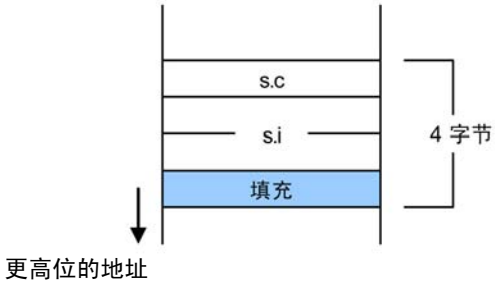


附图 D.4 结构的布局范例 (1)

一般上，结构是没有字对齐的。结构成员会被连续映射。若要使用字对齐，则须使用 #pragma STRUCT 扩展函数。若成员的总大小是奇数， #pragma STRUCT 将添加一字节的填充。附图 D.5 中是一个映射范例。

（范例）

```
#pragma STRUCT  TAG  unpac

struct TAG{
        char    c;
        int     i;
        s;
```



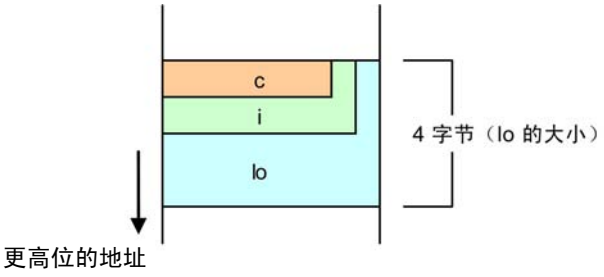
附图 D.5 结构的布局范例 (2)

附录 D.1.7 联合

联合将占用相等于其成员的最大数据大小的区域。附图 D.6 中是一个映射范例。

（范例）

```
union TAG {
        char    c;
        int     i;
        long    lo;
        s;
```



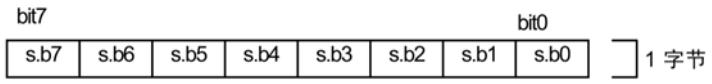
附图 D.6 联合的布局范例

附录 D.1.8　　位字段类型

位字段类型将映射到最低有效位。附图 D.7 中是一个映射范例。

（范例）

```
struct BTAG {
        char    b0 : 1;
        char    b1 : 1;
        char    b2 : 1;
        char    b3 : 1;
        char    b4 : 1;
        char    b5 : 1;
        char    b6 : 1;
        char    b7 : 1;
        } s;
```

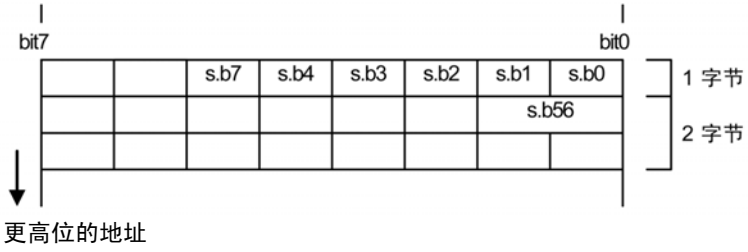


附图 D.7　　位字段的布局范例 (1)

若其中一个位字段成员属于不同的数据类型，那么它将被映射到下一个地址。因此，具有相同数据类型的成员，将从最低地址开始连续映射到该数据类型所将映射到的地址上。

（范例）

```
struct BTAG {
        char    b0 : 1;
        char    b1 : 1;
        char    b2 : 1;
        char    b3 : 1;
        char    b4 : 1;
        int     b56 :2;
        char    b7 : 1;
        } s;
```



附图 D.8　　位字段的布局范例 (2)

注意：　1. 若未指定符号，默认的位字段成员类型将是 unsigned。
　　2. 无法为位字段指定 long long 类型。

附录 D.2 符号的扩展规则

在 ANSI 及其它标准的 C 语言指定下，char 类型数据的符号将被扩展为 int 类型的数据，以便进行计算。这项指定可避免因 char 类型的最大值超出范围，而在执行附图 D.9 中所示的 char 类型计算时返回未预期的结果。

```
void    func(void)
{
        char    c1, c2, c3;

        c1 = c2 * 2 / c3;
}
```

附图 D.9 C 程序的范例

若要生成可最大程度提高代码效率和执行速度的代码， NC30 在默认情况下是不会将 char 类型扩展为 int 类型的。然而，通过使用 nc30 编译驱动器命令行选项 -fansi 或 -fextend_to_int (-fETI) 则可盖写默认设置，从而达到和标准 C 中相同的符号扩展。

若您未使用 -fansi 或 -fextend_to_int (-fETI) 选项，同时您的程序将计算结果指定为 char 类型，如附图 D.9 中所示，则请确保 char 类型的最大值或最小值[1] 不会造成计算溢出。

[1] 在 NC30 中，可表达为 char 类型的值范围如下：
* unsigned char 类型 0. 255,
* signed char 类型 ..... -128. 127

附录 D.3 函数调用的规则

附录 D.3.1 返回值的规则

当从函数返回值时，系统将使用寄存器来返回整数、指针和浮点类型的值。附表 D.4 中显示有关返回值的调用规则。

附表 D.4 返回值相关的调用规则

返回值的类型	规则
_Bool char	R0L 寄存器
int near 指针	R0 寄存器
float long far 指针	通过存储在 R0 寄存器中返回最低有效 16 位。通过存储在 R2 寄存器中返回最高有效 16 位。
double long double	当返回值时，将按 R3、 R2、 R1 和 R0 的顺序，依序存储在从高阶位开始的 16 位中。
long long	当返回值时，将按 R3、 R2、 R1 和 R0 的顺序，依序存储在从高阶位开始的 16 位中。
结构类型 联合类型	调用函数之前，将存储返回值区域的 far 地址保存到堆栈。从调用目标函数返回前，函数会将返回值写入到保存在堆栈的 far 地址所指定的区域。

附录 D.3.2 参数转移的规则

NC30 使用寄存器或堆栈来将参数传递到函数。

(1) 通过寄存器传递参数

当符合下列条件时，系统将使用附表 D.5 和附表 D.6 中所列出的对应的 “Registers Used”（所使用的寄存器）来传递参数。

- 函数进行了原型声明[2]，同时参数的类型在调用函数时已确认。
- 变量参数 “…” 在原型声明中不被使用。
- 有关函数的参数的类型，可参照附表 D.5 和附表 D.6 中的参数和参数类型。

附表 D.5 参数通过寄存器转移的规则 (NC308)

参数	第一个参数	所使用的寄存器
第一个参数	char 类型， _Bool 类型	R0L 寄存器
	int 类型 near 指针类型	R0 寄存器

2. 在输入原型声明时 （即在编写新格式时）， NC30 使用仅限寄存器转移的方式。之后，当输入了 K&R 格式的描述时 （旧格式的描述），所有参数将通过堆栈传递。
另外也请注意，当特定语句中同时存在为函数指定原型声明的描述格式 （新格式）和 K&R 格式的描述 （旧格式）时，由于 C 语言指定的缘故，系统可能无法将参数正确传递到函数。
因此，在为 NC30 编写 C 语言的源文件时，我们建议将原型声明的描述格式用作标准格式。

附表 D.6　　参数通过寄存器转移的规则 (NC30)

参数	第一个参数	所使用的寄存器
第一个参数	char 类型， _Bool 类型	R1L 寄存器
	int 类型 near 指针类型	R1 寄存器
第二个参数	int 类型 near 指针类型	R2 寄存器

(2)　通过堆栈传递参数

所有不符合寄存器转移要求的参数将通过堆栈传递。附表 D.7 和附表 D.8 概括了用来传递参数的方法。

附表 D.7　　将参数传递到函数的规则 (NC308)

参数类型	第一个参数	第二个参数	第三个和之后的参数
char 类型 _Bool 类型	R0L 寄存器	堆栈	堆栈
int 类型 near 指针类型	R0 寄存器	堆栈	堆栈
其它类型	堆栈	堆栈	堆栈

附表 D.8　　将参数传递到函数的规则 (NC30)

参数类型	第一个参数	第二个参数	第三个和之后的参数
char 类型 _Bool 类型	R1L 寄存器	堆栈	堆栈
int 类型 near 指针类型	R1 寄存器	R2 寄存器	堆栈
其它类型	堆栈	堆栈	堆栈

附录 D.3.3　　将函数转换为汇编语言符号的规则

函数在 C 语言源文件中定义的函数名称，在汇编器源文件中用作函数的起始标号。

函数在汇编器源文件中的起始标号，由 C 语言源文件中的函数名称，加上 _（下划线）或 $（货币符号）的前缀组成。

下表显示添加到函数名称的字符串，及添加这些字符串的条件。

附表 D.9　　将字符串添加到函数的条件

所添加的字符串	条件
$（货币符号）	参数中有任何一个通过寄存器传递的函数
_（下划线）	不符合上述条件的函数 [3]

3. 然而，使用 #pragma INTCALL 指定的函数的名称将不会被输出。

附图 D.10 中所示的是函数具有寄存器参数，且函数仅通过堆栈传递其参数的样品程序。

```
int     func_proto( int , int , int);                   [1]

int     func_proto(int i, int j, int k)                 [2]
{
        return i + j + k;
}

int     func_no_proto( i, j, k)                         [3]
int     i;
int     j;
int     k;
{
        return i + j + k;
}

void    main(void)                                      [4]
{
        int     sum;

        sum = func_proto(1,2,3);                        [5]
        sum = func_no_proto(1,2,3);                     [6]
}
```

[1] 这是函数 func_proto 的原型声明。
[2] 这是函数 func_proto 的主体。（由于已输入原型声明，所以这里是新的格式。）
[3] 这是函数 func_no_proto 的主体。（这是 K&R 格式的描述，即旧格式。）
[4] 这是函数 main 的主体。
[5] 这将调用函数 func_proto。
[6] 这将调用函数 func_no_proto。

附图 D.10　　调用函数的样品程序 (sample.c)

上述样品程序的编译结果在下一页显示。附图 D.11 中显示程序第 [2] 部分定义函数 func_proto 的编译结果。附图 D.12 中显示程序第 [3] 部分定义函数 func_no_proto 的编译结果。附图 D.13 显示程序第 [4] 部分调用函数 func_proto 和函数 func_no_proto 的编译结果。

```
;## #     FUNCTION func_proto
;## #     FRAME     AUTO      (        j)  size  2,     offset -4
;## #     FRAME     AUTO      (        i)  size  2,     offset -2
;## #  FRAME   ARG (        k) size   2,        offset 5                    ←[7]
;## #  REGISTER ARG   (      i)  size     2,      REGISTER R1                ←[8]
;## #  REGISTER ARG   (      j)  size     2,      REGISTER R2                ←[9]
;## #     ARG Size(2)           Auto Size(2)            Context Size(5)


          .SECTION  program,CODE,ALIGN
          ._file    'sample.c'
          .align
          ._line    4
;## # C_SRC :      {
          .glb      $func_proto
$func_proto:                                                               ←[10]
          enter     #04H
          mov.w     R1,-2[FB]   ;   i  i
          mov.w     R2,-4[FB]   ;   j  j
          ._line    5
;## # C_SRC :                    return i + j + k;
          mov.w     -2[FB],R0   ;   i
          add.w     -4[FB],R0   ;   j
          add.w     5[FB],R0    ;   k
          exitd
E1:
```

[7] 这将通过堆栈传递第三个参数 k。
[8] 这将通过寄存器传递第二个参数 i。
[9] 这将通过寄存器传递第一个参数 j。
[10] 这是函数 func_proto 的起始地址。

附图 D.11　样品程序的编译结果 (sample.c) (1)

在附图 D.10 中所列的样品程序 (sample.c) 的编译结果 (1) 中，由于函数 func_proto 进行了原型声明，因此第一个和第二个参数通过寄存器传递。由于第三个参数不受寄存器转移的影响，因此它通过堆栈传递。

此外，由于函数的参数通过寄存器传递，因此函数起始地址的符号名称，由在 C 语言源文件中描述的“func_proto”加上 $ （货币符号）的前缀取得，即 “$func_proto”。

```
;## #     FUNCTION func_no_proto
;## # FRAME   ARG   (          i) size  2,     offset 5                    [11]
;## # FRAME   ARG   (          j) size  2,     offset 7
;## # FRAME   ARG   (          k) size  2,     offset 9
;## #     ARG Size(6)          Auto Size(0)          Context Size(5)

            .align
            ._line          12
;## # C_SRC :              {
            .glb            _func_no_proto                          ←[12]
_func_no_proto:
            enter           #00H
            ._line          13
;## # C_SRC :                      return i + j + k;
            mov.w           5[FB],R0  ;  i
            add.w           7[FB],R0  ;  j
            add.w           9[FB],R0  ;  k
            exitd
E2:
```

[11] 这将通过堆栈传递所有参数。
[12] 这是函数 func_no_proto 的起始地址。

附图 D.12　　样品程序的编译结果 (sample.c) (2)

在附图 D.10 中所列的样品程序 (sample.c) 的编译结果 (2) 中，由于函数 func_no_proto 以 K&R 格式编写，因此所有参数通过堆栈传递。

此外，由于函数的参数未通过寄存器传递，因此函数起始地址的符号名称，由在 C 语言源文件中描述的“func_no_proto”加上 _（下划线）的前缀取得，即 “_func_no_proto”。

```
;## #     FUNCTION main
;## #     FRAME    AUTO       (          sum) size  2,     offset -2
;## #     ARG Size(0)          Auto Size(2)               Context Size(5)

          .align
          ._line     17
;## # C_SRC :       {
          .glb       _main
_main:
          enter      #02H
          ._line     20
;## # C_SRC :                  sum = func_proto(1,2,3);
          push.w     #0003H                                          [13]
          mov.w      #0002H,R2
          mov.w      #0001H,R1
          jsr        $func_proto
          add.l      #02H,SP
          mov.w      R0,-2[FB]  ; _sum
          ._line     21
;## # C_SRC :                  sum = func_no_proto(1,2,3);
          push.w     #0003H                                          [14]
          push.w     #0002H
          push.w     #0001H
          jsr        _func_no_proto
          add.l      #06H,SP
          mov.w      R0,-2[FB]  ; _sum
          ._line     22
;## # C_SRC :       }
          exitd
E3:
          .END
```

附图 D.13　　样品程序的编译结果 (sample.c) (3)

在附图 D.13 中，第 [13] 部分调用 func_proto，而第 [14] 部分则调用 func_no_proto。

附录 D.3.4 函数之间的连接

附图 D.16 到附图 D.18 中显示附图 D.14 中所示程序的堆栈帧构建和释放处理。附图 D.15 中显示当对附图 D.14 中所示的程序进行编译后所产生的汇编语言程序。

```
int     func( int, int ,int);

void    main(void)
{
        int             i = 0x1234;         ← func 的参数
        int             j = 0x5678;         ← func 的参数
        int             k = 0x9abc;         ← func 的参数

        k = func( i, j ,k);
}

int     func( int x,int y,int z )
{
        int             sum;

        sum = x + y + z ;
        return sum;                         ← 返回值到 main
}
```

附图 D.14 C 语言样品程序的范例

```
;## #     FUNCTION   main
;## #     FRAME      AUTO     (          i)  size  2,     offset -6
;## #     FRAME      AUTO     (          j)  size  2,     offset -4
;## #     FRAME      AUTO     (          k)  size  2,     offset -2
;## #     ARG Size(0)         Auto Size(6)               Context Size(5)


          .SECTION   program,CODE,ALIGN
          ._file     'sample.c'
          .align
          ._line     4
;## # C_SRC :        {
          .glb       _main
_main:                                                              ← [1]
          enter      #06H                                           ← [2]
          ._line     5
;## # C_SRC :                int          i = 0x1234;
          mov.w      #1234H,-2[FB]        ; i
          ._line     6
;## # C_SRC :                int          j = 0x5678;
          mov.w      #5678H,-4[FB]        ; j
          ._line     7
;## # C_SRC :                int          k = 0x9abc;
          mov.w      #9abcH,-6[FB]        ; k
          ._line     9
;## # C_SRC :                k = func( i, j ,k);
          push.w     -6[FB]     ; k                                 ← [3]
          mov.w      -4[FB],R2  ; j                                 ← [4]
          mov.w      -2[FB],R1  ; i                                 ← [5]
          jsr        $func                                          ← [6]
          add.l      #02H,SP                                        ← [10]
          mov.w      R0,-2[FB]  ; k                                 ← [11]
          ._line     10
;## # C_SRC :        }
          exitd
E1:
```

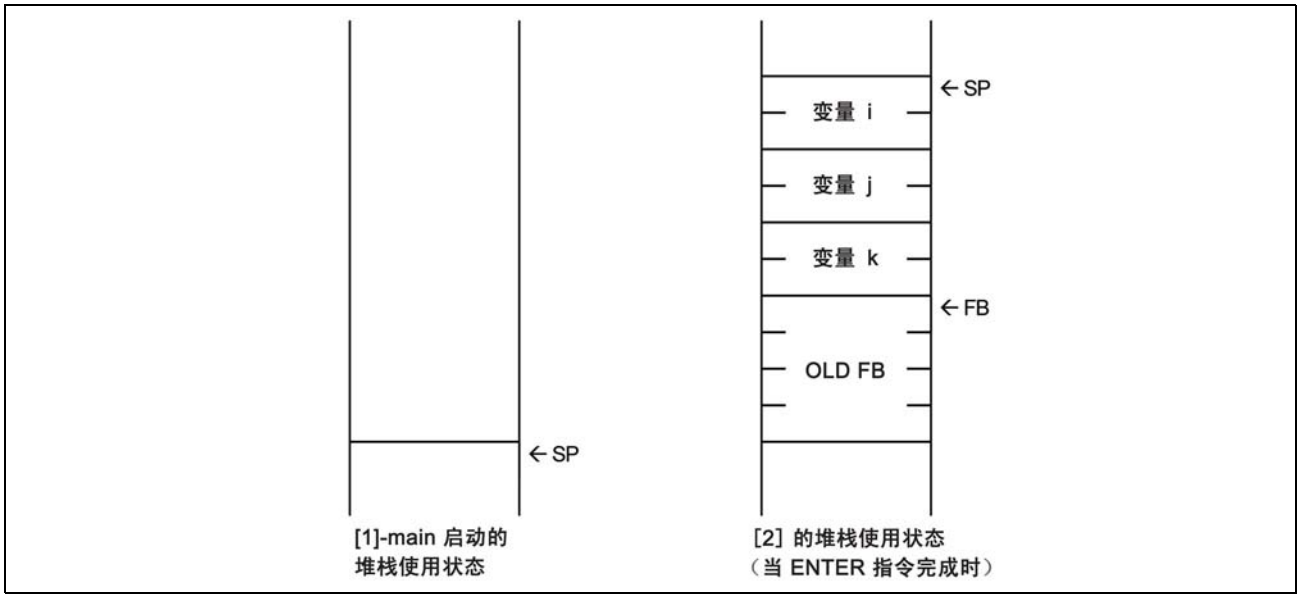
附图 D.15　　汇编语言样品程序 (1/2)

```
;## #     FUNCTION func
;## #     FRAME     AUTO     (        sum)  size     2,         offset -4
;## #     FRAME     AUTO     (          y)  size     2,         offset -4
;## #     FRAME     AUTO     (          x)  size     2,         offset -2
;## #  FRAME    ARG    (         z)  size   2,     offset 5
;## #  REGISTER ARG     (         x)  size     2,         REGISTER R1
;## #  REGISTER ARG     (         y)  size     2,         REGISTER R2
;## #     ARG Size(2)              Auto Size(4)                Context Size(5)

          .align
          ._line       13
;## # C_SRC :       {
          .glb         $func
$func:
          enter        #04H                                              ← [7]
          mov.w        R1,-2[FB]    ;   x  x
          mov.w        R2,-4[FB]    ;   y  y
          ._line       16
;## # C_SRC :                      sum = x + y + z ;
          mov.w        -2[FB],R0    ;   x
          add.w        -4[FB],R0    ;   y
          add.w        5[FB],R0     ;   z
          mov.w        R0,-4[FB]    ;   sum
          ._line       17
;## # C_SRC :                      return sum;
          mov.w        -4[FB],R0    ;   sum                           ← [8]
          exitd                                                          ← [9]
E2:
          .END
```

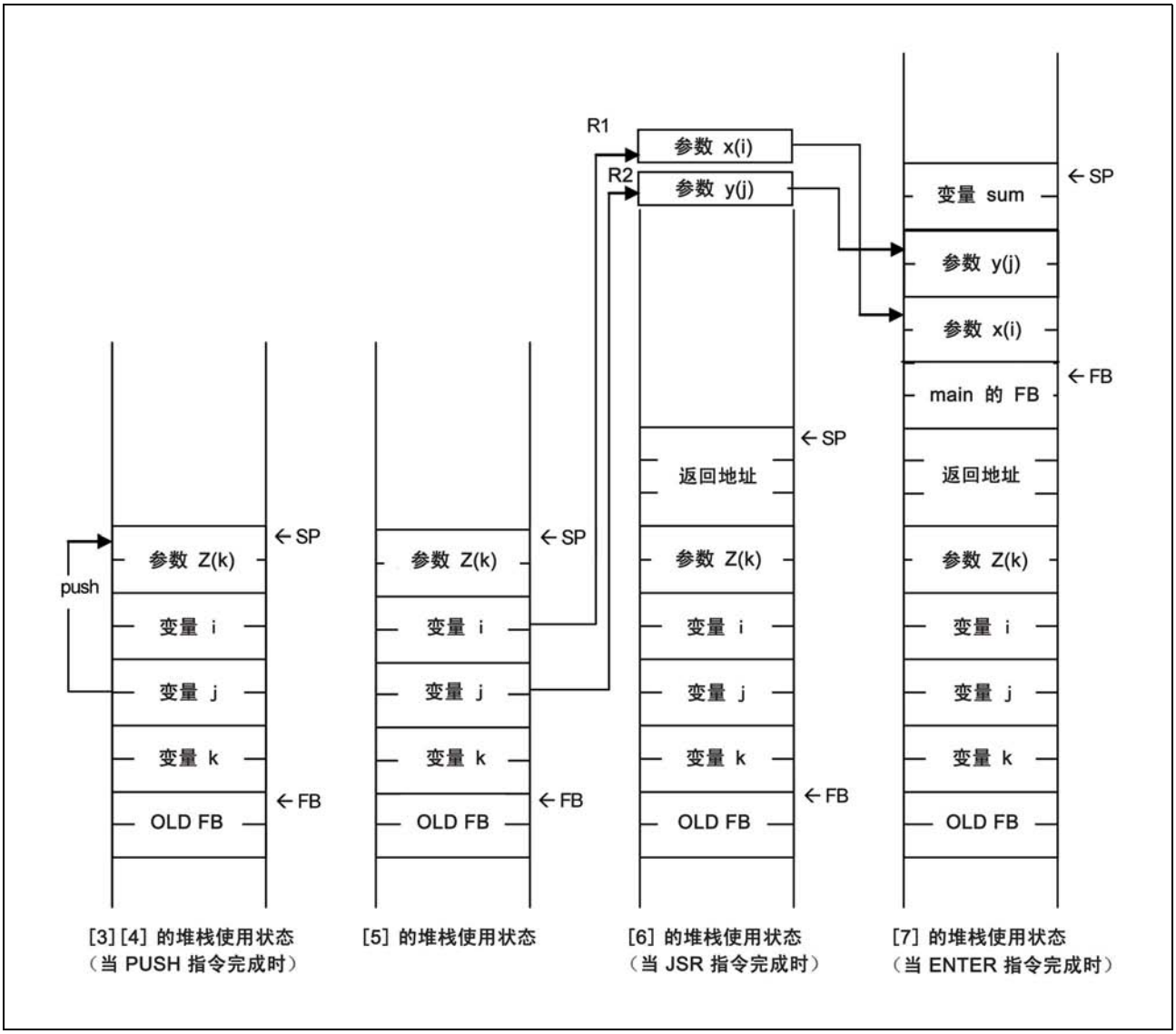
附图 D.16　　汇编语言样品程序 (2/2)

下面附图 D.17 到附图 D.19 显示了附图 D.15 中各项处理的堆栈和寄存器转移。[1] → [2] 中的处理 （函数 main 的进入处理）在附图 D.17 中显示。[3] → [4] → [5] → [6] → [7] 的处理 （调用函数 func 和构建函数 func 中所使用堆栈帧的处理）在附图 D.18 中显示。

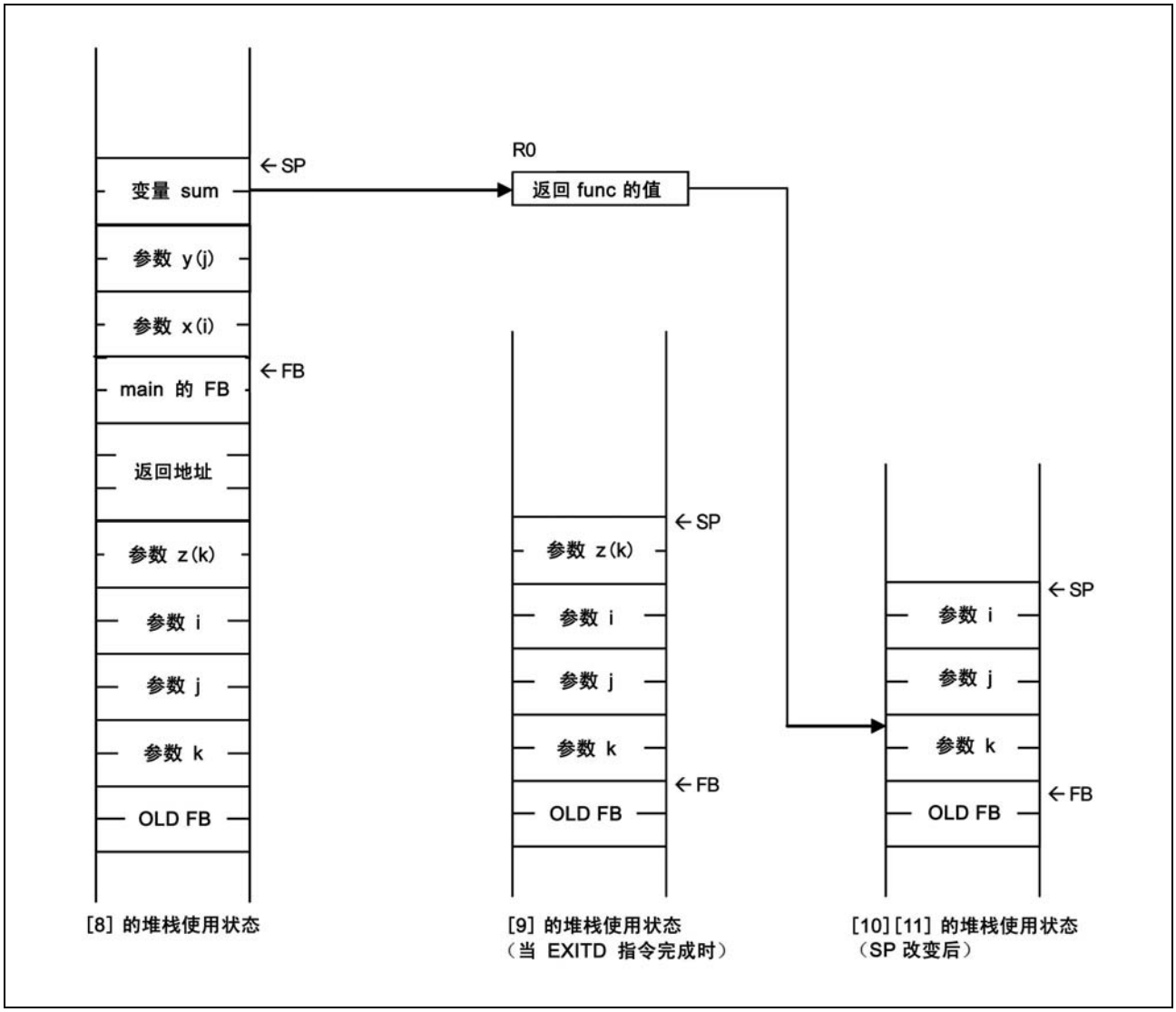
[8] → [9] → [10] → [11] 的处理 （从函数 func 返回到函数 main 的处理）在附图 D.19 中显示。



附图 D.17　　函数 main 的进入处理



附图 D.18　　调用函数 func 和进入处理



附图 D.19　　函数 func 的退出处理

附录 D.4　保留 auto 变量的区域

存储类为 auto 的变量将存储在单片机的堆栈中。对于类似附图 D.20 中所示的 C 语言源文件，若存储类为 auto 的变量可使用的区域不互相重叠，系统将只分配一个区域，然后由多个变量共享。

```
void    func(void)
{
        int             i, j, k;

    for( i=0 ; i<=0 ; i++ ){
                process                                 i 的范围
    }
                :
            ( 已省略 )
                :
    for( j=0xFF ; j<=0 ; j-- ){
                process                                 j 的范围
    }
                :
            ( 已省略 )
                :
    for( k=0 ; k<=0 ; k++ ){
                process                                 k 的范围
    }
}
```

附图 D.20　C 程序的范例

在本范例中，三个 auto 变量 i、 j 和 k 的有效范围不互相重叠，因此它们将共享两字节的区域 （从 FB 偏移 1）。附图 D.21 中显示编译附图 D.20 中所示的程序而生成的汇编语言源文件。

```
;### FUNCTION      func
;###      FRAME AUTO      (       k)      size 2,         offset -2               ← [1]
;###      FRAME AUTO      (       j)      size 2,         offset -2               ← [2]
;###      FRAME AUTO      (       i)      size 2,         offset -2               ← [3]
        .section  program
        ._file          'test1.c'
        ._line          3
        .glb            _func
_func:
        enter           #02H
                :
        （其余省略）


* 如 [1]、 [2] 和 [3] 所示，三个 auto 变量共享 FB 偏移 -2 的区域。
```

附图 D.21　汇编语言源程序的范例

附录 D.5 寄存器的转义规则

寄存器在调用 C 函数时的转义规则如下：

1. 寄存器在调用 C 函数时的转义规则如下：
   - 在 C 调用目标函数中使用的寄存器
2. 应该在调用目标函数的进入程序转义的寄存器。
   - 无

附录 E　标准程序库

附录 E.1　　标准标题文件

当使用 NC30 标准程序库时，必须包含定义该函数的标题文件。

本附录将详细介绍标准 NC30 标题文件的函数及其规格说明。

附录 E.1.1　　标准标题文件的内容

NC30 包含附表 E.1 中所示的 15 个标准标题文件。

附表 E.1　　标准标题文件列表

标题文件的名称	内容
assert.h	输出程序的诊断信息。
ctype.h	将字符确定函数声明为宏。
errno.h	定义错误编号。
float.h	定义各种与浮点的内部表达相关的限制值。
limits.h	定义各种与编译器的内部处理相关的限制值。
locale.h	定义 / 声明操作程序的地域化的宏和函数。
math.h	声明内部处理的算术 / 逻辑函数。
mathf.h	声明 （float 类型）内部处理的算术 / 逻辑函数。
setjmp.h	定义在转移函数中使用的结构。
signal.h	定义 / 声明所需的异步中断处理。
stdarg.h	定义 / 声明实际参数的数量为变量的函数。
stddef.h	定义在标准包含文件之间共享的宏的名称。
stdio.h	1. 定义 FILE 结构。 2. 定义流名称。 3. 声明 I/O 函数的原型。
stdlib.h	声明存储器管理的原型，并终止函数。
string.h	声明字符串和存储器处理函数的原型。
time.h	声明表示当前日历时间所需的函数，并定义其类型。

附录 E.1.2 标准标题文件参考

下面是 NC30 随附的标准标题文件的详细描述。标题文件将按字母顺序呈现。

标题文件中所声明的 NC30 标准函数和定义数据类型的数值表达式限制的宏，分别在有关的标题文件中进行描述。

assert.h

功能： 定义 assert 函数。

ctype.h

功能： 定义 / 声明字符串处理函数。下面所列出的是字符串处理函数。

函数	内容
isalnum	确认字符是字母，还是数字。
isalpha	确认字符是否是字母。
iscntrl	确认字符是否是控制字符。
isdigit	确认字符是否是数字。
isgraph	确认字符是否可打印 （除了空白符）。
islower	确认字符是否是小写字母。
isprint	确认字符是否可打印 （包括空白符）。
ispunct	确认字符是否是标点字符。
isspace	确认字符是空白符、制表符，还是新行。
isupper	确认字符是否是大写字母。
isxdigit	确认字符是否是十六进制字符。
tolower	将字符从大写转换为小写。
toupper	将字符从小写转换为大写。

errno.h

功能： 定义错误编号。

float.h

功能： 定义浮点值的内部表达的限制。下面列出了定义浮点值的限制的宏。
在 NC30 中， long double 类型被作为 double 类型处理。因此，适用于 double 类型的限制，同时也适用于 long double 类型。

宏的名称	内容	所定义的值
DBL_DIG	double 类型十进制精度的最大位数	15
DBL_EPSILON	1.0+DBL_EPSILON 不等于 1.0 的最小正数值	2.2204460492503131e-16
DBL_MANT_DIG	在表达中使 double 类型浮点值与基数匹配时，其尾数部分的最大位数	53
DBL_MAX	double 类型变量所能接受的最大值	1.7976931348623157e+30
DBL_MAX_10_EXP	可以使用 double 类型的浮点数值来表示的最大 10 次幂值	30
DBL_MAX_EXP	可以使用 double 类型的浮点数值来表示的最大基数幂值	1024
DBL_MIN	double 类型变量所能接受的最小值	2.2250738585072014e-30
DBL_MIN_10_EXP	可以使用 double 类型的浮点数值来表示的最小 10 次幂值	-307
DBL_MIN_EXP	可以使用 double 类型的浮点数值来表示的最小基数幂值	-1021
FLT_DIG	float 类型十进制精度的最大位数	6
FLT_EPSILON	1.0+FLT_EPSILON 不等于 1.0 的最小正数值	1.19209290e-07F
FLT_MANT_DIG	在表达中使 float 类型浮点值与基数匹配时，其尾数部分的最大位数	24
FLT_MAX	float 类型变量所能接受的最大值	3.40282347e+38F
FLT_MAX_10_EXP	可以使用 float 类型的浮点数值来表示的最大 10 次幂值	38
FLT_MAX_EXP	可以使用 float 类型的浮点数值来表示的最大基数幂值	128
FLT_MIN	float 类型变量所能接受的最小值	1.17549435e-38F
FLT_MIN_10_EXP	可以使用 float 类型的浮点数值来表示的最小 10 次幂值	-37
FLT_MIN_EXP	可以使用 float 类型的浮点数值来表示的最大基数幂值	-125
FLT_RADIX	浮点表达中的指数基数	2
FLT_ROUNDS	舍入浮点数字的方法	1 （整数就近舍入）

limits.h

功能： 定义适用于编译器的内部处理的限制。下面列出了定义这些限制的宏。

宏的名称	内容	所定义的值
MB_LEN_MAX	多字节字符类型字节的最大值	1
CHAR_BIT	char 类型位的数量	8
CHAR_MAX	char 类型变量所能接受的最大值	255
CHAR_MIN	char 类型变量所能接受的最小值	0
SCHAR_MAX	signed char 类型变量所能接受的最大值	127
SCHAR_MIN	signed char 类型变量所能接受的最小值	-128
INT_MAX	int 类型变量所能接受的最大值	32767
INT_MIN	int 类型变量所能接受的最小值	-32768
SHRT_MAX	short int 类型变量所能接受的最大值	32767
SHRT_MIN	short int 类型变量所能接受的最小值	-32768
LONG_MAX	long 类型变量所能接受的最大值	2147483647
LONG_MIN	long 类型变量所能接受的最小值	-2147483648
LLONG_MAX	long long 类型变量所能接受的最大值	9223372036854775807
LLONG_MIN	long long 类型变量所能接受的最小值	-9223372036854775808
UCHAR_MAX	unsigned char 类型变量所能接受的最大值	255
UINT_MAX	unsigned int 类型变量所能接受的最大值	65535
USHRT_MAX	unsigned short int 类型变量所能接受的最大值	65535
ULONG_MAX	unsigned long int 类型变量所能接受的最大值	4294967295
ULLONG_MAX	unsigned long long int 类型变量所能接受的最大值	18446744073709551615

locale.h

功能： 定义 / 声明操作程序的地域化的宏和函数。下面是区域函数列表。

函数	内容
localeconv	初始化 struct lconv。
setlocale	设置及搜索程序的区域信息。

math.h

功能： 声明数学函数的原型。下面是数学函数列表。

函数	内容
acos	计算反余弦值。
asin	计算反正弦值。
atan	计算反正切值。
atan2	计算反正切值。
ceil	计算整数进位值。
cos	计算余弦值。
cosh	计算双曲余弦值。
exp	计算指数函数。
fabs	计算双精度浮点数的绝对值。
floor	计算整数借位值。
fmod	计算余数。
frexp	将浮点数分为尾数部分和指数部分。
labs	计算 long 类型整数的绝对值。
ldexp	计算浮点数的幂。
log	计算自然对数。
log10	计算常用对数。
modf	将实数分为尾数部分和指数部分的计算。
pow	计算数值的幂。
sin	计算正弦值。
sinh	计算双曲正弦值。
sqrt	计算数值的平方根。
tan	计算正切值。
tanh	计算双曲正切值。

setjmp.h

功能：　定义在转移函数中使用的结构。

函数	内容
longjmp	执行全局跳转。
setjmp	为全局跳转设置一个堆栈环境。

signal.h

功能：　定义 / 声明所需的异步中断处理。

stdarg.h

功能：　定义 / 声明实际参数的数量为变量的函数。

stddef.h

功能：　定义在标准包含文件之间共享的宏的名称。

stdio.h

功能：　定义 FILE 结构、流名称，及声明 I/O 函数原型。这将对下列函数进行原型声明。

类型	函数	功能
初始化	init	初始化 M16C/80 族的输入 / 输出。
	clearerr	初始化 （清除）错误状态说明符。
输入	fgetc	从流输入一个字符。
	getc	从流输入一个字符。
	getchar	从 stdin 输入一个字符。
	fgets	从流输入一个行。
	gets	从 stdin 输入一个行。
	fread	从流输入特定的数据项目。
	scanf	从 stdin 输入带格式的字符。
	fscanf	从流输入带格式的字符。
	sscanf	从字符串输入带格式的数据。
输出	fputc	输出一个字符到流。
	putc	输出一个字符到流。
	putchar	输出一个字符到 stdout。
	fputs	输出一个行到流。
	puts	输出一个行到 stdout。
	fwrite	输出特定的数据项目到流。
	perror	输出错误信息到 stdout。
	printf	输出带格式的字符到 stdout。
	fflush	清除输出缓冲器的流。
	Fprintf	输出带格式的字符到流。
	sprintf	写入带格式的文本到字符串。
	vfprintf	对流进行带格式的输出。
	vprintf	对 stdout 进行带格式的输出。
	vsprintf	对缓冲器进行带格式的输出。
返回	ungetc	将一个字符发送到输入流。
确定	ferror	确认输入 / 输出错误。
	feof	确认 EOF （文件末尾）。

stdlib.h

功能： 声明存储器管理的原型，并终止函数。

功能	内容
abort	终止执行程序。
abs	计算一个整数的绝对值。
atof	将字符串转换为一个 double 类型浮点数。
atoi	将字符串转换为一个 int 类型整数。
atol	将字符串转换为一个 long 类型整数。
bsearch	对一个数组执行二进制搜索。
calloc	分配一个存储区，并将它初始化为零 (0) 。
div	对一个 int 类型整数执行除法，并计算余数。
free	释放所分配的存储区。
labs	计算 long 类型整数的绝对值。
ldiv	对一个 long 类型整数执行除法，并计算余数。
malloc	分配存储区。
mblen	计算多字节字符串的长度。
mbstowcs	将多字节字符串转换为一个宽字符串。
mbtowc	将多字节字符转换为一个宽字符。
qsort	为数组中的元素进行排序。
realloc	更改所分配的一个存储区的大小。
strtod	将字符串转换为一个 double 类型整数。
strtol	将字符串转换为一个 long 类型整数。
strtoul	将字符串转换为一个 unsigned long 类型整数。
wcstombs	将宽字符串转换为一个多字节字符串。
wctomb	将宽字符转换为一个多字节字符。

string.h

功能： 声明字符串处理函数及存储器处理函数的原型。

类型	类型	内容
复制	strcpy	复制一个字符串。
	strncpy	复制一个字符串 （‘n’字符）。
连接	strcat	连接字符串。
	strncat	连接字符串 （‘n’字符）。
比较	strcmp	比较字符串。
	strcoll	比较字符串 （使用区域信息）。
	stricmp	比较字符串。（所有字母作为大写字母处理。）
	strncmp	比较字符串 （‘n’字符）。
	strnicmp	比较字符串 （‘n’字符）。（所有字母作为大写字母处理。）
搜索	strchr	从字符串顶端开始搜索特定的字符。
	strcspn	计算未在另一字符串中找到的未指定字符的长度 （字数）。
	strpbrk	从另一字符串搜索字符串中的特定字符。
	strrchr	从字符串的末尾搜索特定字符。
	strspn	计算在另一字符串中找到的指定字符的长度 （字数）。
	strstr	从字符串搜索特定字符。
	strtok	在一个字符串中将其中一些字符串划分为标志。
长度	strlen	计算一个字符串中的字符数。
转换	strerror	将错误编号转换为一个字符串。
	strxfrm	转换一个字符串 （使用区域信息）。
初始化	bzero	初始化一个存储区 （将它清零）。
复制	bcopy	从一个存储区复制字符到另一个存储区。
	memcpy	从一个存储区复制字符 （‘n’字节）到另一个存储区。
	memset	通过填充字符的方式来设置一个存储区。
比较	memcmp	比较存储区 （‘n’字节）。
	memicmp	比较存储区 （将字母作为大写字母处理）。
搜索	memchr	从存储区搜索一个字符。

time.h

功能： 声明表示当前日历时间所需的函数，并定义其类型。

附录 E.2 标准函数参考

描述编译器标准函数程序库的功能及指定详情。

附录 E.2.1 标准程序库概述

NC30 具有 119 个标准程序库项目。根据功能，每个函数可被分配到下列 11 个类别中。

1. 字符串处理函数
   用于复制及比较字符串等的函数。
2. 字符处理函数
   用于判断字母和十进制字符等，及将大写转换为小写或反之的函数。
3. I/O 函数
   输入及输出字符和字符串的函数。这包括用于带格式的 I/O 和字符串操作的函数。
4. 存储器管理函数
   用于动态保留及释放存储区的函数。
5. 存储器操作函数
   用于复制、设置及比较存储区的函数。
6. 指定控制函数
   用于执行和终止程序，及从当前执行函数跳转到另一函数的函数。
7. 数学函数
   * 这些函数需要较多时间来执行。
   - 因此，请留意看门狗定时器的使用。
8. 整数算术函数
   用于对整数值执行计算的函数。
9. 转换字符串的值的函数
   用于将字符串转换为数值的函数。
10. 多字节字符和多字节字符串操作函数
    用于处理多字节字符和多字节字符串的函数。
11. 区域函数
    区域相关的函数。

附录 E.2.2 标准程序库函数按函数区分的列表

(a) 字符串处理函数

下面是字符串处理函数的列表。

附表 E.2 字符串处理函数

类型	函数	内容	可重入
复制	strcpy	复制一个字符串。	O
	strncpy	复制一个字符串 （‘n’字符）。	O
连接	strcat	连接字符串。	O
	strncat	连接字符串 （‘n’字符）。	O
比较	strcmp	比较字符串。	O
	strcoll	比较字符串 （使用区域信息）。	O
	stricmp	比较字符串。（所有字母作为大写字母处理。）	O
	strncmp	比较字符串 （‘n’字符）。	O
	strnicmp	比较字符串 （‘n’字符）。（所有字母作为大写字母处理。）	O
搜索	strchr	从字符串顶端开始搜索特定的字符。	O
	strcspn	计算未在另一字符串中找到的未指定字符的长度 （字数）。	O
	strpbrk	从另一字符串搜索字符串中的特定字符。	O
	strrchr	从字符串的末尾搜索特定字符。	O
	strspn	计算在另一字符串中找到的指定字符的长度 （字数）。	O
	strstr	从字符串搜索特定字符。	O
	strtok	在一个字符串中将其中一些字符串划分为标志。	×
长度	strlen	计算一个字符串中的字符数。	O
转换	strerror	将错误编号转换为一个字符串。	×
	strxfrm	转换一个字符串 （使用区域信息）。	O

* 一些标准函数会使用该函数专用的全局变量。若在调用及执行该函数的过程中发生了中断，中断处理程序将调用相同函数，而第一次调用函数时所使用的全局变量将可能被盖写。
这不会发生在具有重入性的全局函数变量 （在表中以 O 表示）。不过，若函数不具有重入性 （在表中以 X 表示），则必须留意函数是否也被中断处理程序所使用。

(b) 字符处理函数

下面是字符处理函数的列表。

附表 E.3 字符处理函数

函数	内容	可重入
isalnum	确认字符是字母，还是数字。	O
isalpha	确认字符是否是字母。	O
iscntrl	确认字符是否是控制字符。	O
isdigit	确认字符是否是数字。	O
isgraph	确认字符是否可打印 （除了空白符）。	O
islower	确认字符是否是小写字母。	O
isprint	确认字符是否可打印 （包括空白符）。	O
ispunct	确认字符是否是标点字符。	O
isspace	确认字符是空白符、制表符，还是新行。	O
isupper	确认字符是否是大写字母。	O
isxdigit	确认字符是否是十六进制字符。	O
tolower	将字符从大写转换为小写。	O
toupper	将字符从小写转换为大写。	O

(c) I/O 函数

下面是 I/O 函数的列表。

附表 E.4 I/O 函数

类型	函数	内容	可重入
初始化	init	初始化 M16C 系列的输入 / 输出。	×
	clearerror	初始化 （清除）错误状态说明符。	×
初始化	fgetc	从流输入一个字符。	×
	getc	从流输入一个字符。	×
	getchar	从 stdin 输入一个字符。	×
	fgets	从流输入一个行。	×
	gets	从 stdin 输入一个行。	×
	fread	从流输入特定的数据项目。	×
	scanf	从 stdin 输入带格式的字符。	×
	fscanf	从流输入带格式的字符。	×
	sscanf	从字符串输入带格式的数据。	×
输出	fputc	输出一个字符到流。	×
	putc	输出一个字符到流。	×
	putchar	输出一个字符到 stdout。	×
	fputs	输出一个行到流。	×
	puts	输出一个行到 stdout。	×
	fwrite	输出特定的数据项目到流。	×
	perror	输出错误信息到 stdout。	×
	printf	输出带格式的字符到 stdout。	×
	fflush	清除输出缓冲器的流。	×
	fprintf	输出带格式的字符到流。	×
	sprintf	写入带格式的文本到字符串。	×
	vfprintf	对流进行带格式的输出。	×
	vprintf	对 stdout 进行带格式的输出。	×
	vsprintf	对缓冲器进行带格式的输出。	×
返回	ungetc	发回一个字符到输入流。	×
确定	ferror	确认输入 / 输出错误。	×
	feof	确认 EOF （文件末尾）。	×

(d) 存储器管理函数

下面是存储器管理函数的列表。

附表 E.5 存储器管理函数

函数	内容	可重入
calloc	分配一个存储区，并将它初始化为零 (0) 。	×
free	释放所分配的存储区。	×
malloc	分配存储区。	×
realloc	更改所分配的一个存储区的大小。	×

(e) 存储器处理函数

下面是存储器处理函数的列表。

附表 E.6 存储器处理函数

类型	函数	内容	可重入
初始化	bzero	初始化一个存储区 （将它清零）。	O
复制	bcopy	从一个存储区复制字符到另一个存储区。	O
	memcpy	从一个存储区复制字符 （‘n’ 字节）到另一个存储区。	O
	memset	通过填充字符的方式来设置一个存储区。	O
比较	memcmp	比较存储区 （‘n’ 字节）。	O
	memicmp	比较存储区 （将字母作为大写字母处理）。	O
移动	memmove	移动字符串的存储区。	O
搜索	memchr	从存储区搜索一个字符。	O

(f) 指定控制函数

下面是执行控制函数的列表。

附表 E.7 指定控制函数

函数	内容	可重入
abort	终止执行程序。	O
longjmp	执行全局跳转。	O
setjmp	为全局跳转设置一个堆栈环境。	O

(g) 数学函数

下面是数学函数的列表。

附表 E.8 数学函数

函数	内容	可重入
acos	计算反余弦值。	O
asin	计算反正弦值。	O
atan	计算反正切值。	O
atan2	计算反正切值。	O
ceil	计算整数进位值。	O
cos	计算余弦值。	O
cosh	计算双曲余弦值。	O
exp	计算指数函数。	O
fabs	计算双精度浮点数的绝对值。	O
floor	计算整数借位值。	O
fmod	计算余数。	O
frexp	将浮点数分为尾数部分和指数部分。	O
labs	计算 long 类型整数的绝对值。	O
ldexp	计算浮点数的幂。	O
log	计算自然对数。	O
log10	计算常用对数。	O
modf	将实数分为尾数部分和指数部分的计算。	O
pow	计算数值的幂。	O
sin	计算正弦值。	O
sinh	计算双曲正弦值。	O
sqrt	计算数值的平方根。	O
tan	计算正切值。	O
tanh	计算双曲正切值。	O

(h) 整数算术函数

下面是整数算术函数的列表。

附表 E.9 整数算术函数

函数	内容	可重入
abs	计算一个整数的绝对值。	O
bsearch	对一个数组执行二进制搜索。	O
div	对一个 int 类型整数执行除法，并计算余数。	O
labs	计算 long 类型整数的绝对值。	O
ldiv	对一个 long 类型整数执行除法，并计算余数。	O
qsort	为数组中的元素进行排序。	O
rand	生成一个伪随机数。	O
srand	将种子给予生成例程的伪随机数。	O

(i) 转换字符串的值的函数

下面是字符串值转换函数的列表。

附表 E.10 转换字符串的值的函数

函数	内容	可重入
atof	将字符串转换为一个 double 类型浮点数。	O
atoi	将字符串转换为一个 int。	O
atol	将字符串转换为一个 long	O
strtod	将字符串转换为一个 double	O
strtol	将字符串转换为一个 long	O
strtou	将字符串转换为一个 unsigned long 类型整数。	O

(j) 多字节字符和多字节字符串操作函数

下面是多字节字符和多字节字符串操作函数的列表。

附表 E.11 多字节字符和多字节字符串操作函数

函数	内容	可重入
mblen	计算多字节字符串的长度。	O
mbstowcs	将多字节字符串转换为一个宽字符串。	O
mbtowc	将多字节字符转换为一个宽字符。	O
wcstombs	将宽字符串转换为一个多字节字符串。	O
wctomb	将宽字符转换为一个多字节字符。	O

(k) 地域化函数

下面是地域化函数的列表。

附表 E.12 地域化函数

函数	内容	可重入
localeconv	初始化 struct lconv。	O
setlocale	设置及搜索程序的区域信息。	O

附录 E.2.3 标准函数参考

下面将描述 NC30 所提供的标准函数的指定详情。函数按字母顺序排列。

当使用函数时，必须包含 “Format”（格式）下所示的标准标题文件 （extension.h）。

A

abort

指定控制函数

功能： 终止异常执行的程序。

格式： #include <stdlib.h>

void abort(void);

方法： 函数

变量： 不使用任何参数。

返回值： 不返回任何值。

描述： 终止异常执行的程序。

注意： 事实上，程序会在 abort 函数中循环。

abs

整数算术函数

功能： 计算一个整数的绝对值。

格式： #include <stdlib.h>

int abs(n);

方法： 函数

变量： int n; 整数

返回值： 返回整数 n 的绝对值 （和 0 的距离）。

acos

数学函数

功能： 计算反余弦值。

格式：
```
#include <math.h>
double acos( x );
```

方法： 函数

变量：

double x;		任意实数

返回值：
- 若特定实数 x 的值超出 -1.0 到 1.0 的范围，将假设发生错误并返回 0。
- 否则，将返回范围介于 0 到 π 弧度之间的值。

asin

数学函数

功能： 计算反正弦值。

格式：
```
#include <math.h>
double asin( x );
```

方法： 函数

变量：

double x;		任意实数

返回值：
- 若特定实数 x 的值超出 -1.0 到 1.0 的范围，将假设发生错误并返回 0。
- 否则，将返回范围介于 -π/2 到 π/2 弧度之间的值。

atan

数学函数

功能： 计算反正切值。

格式：
```
#include <math.h>
double atan( x );
```

方法： 函数

变量：

double x;		任意实数

返回值： 这将返回范围介于 -π/2 到 π/2 弧度之间的值。

atan2

数学函数

功能： 计算反正切值。

格式： #include <math.h>

double atan2(x , y);

方法： 函数

变量： double x; 任意实数
double y; 任意实数

返回值： 这将返回范围介于 -π 到 π 弧度之间的值。

atof

转换字符串的值的函数

功能： 将字符串转换为一个 double 类型浮点数。

格式： #include <stdlib.h>

double atof(s);

方法： 函数

变量： const char _far *s; 转换字符串的指针

返回值： 返回将字符串转换为双精度浮点数所产生的值。

atoi

字符串转换函数

功能： 将字符串转换为一个 int 类型整数。

格式： #include <stdlib.h>

int atoi(s);

方法： 函数

变量： const char _far *s; 转换字符串的指针

返回值： 返回将字符串转换为一个 int 类型整数所产生的值。

atol

字符串转换函数

功能： 将字符串转换为一个 long 类型整数。

格式：
```
#include <stdlib.h>
long atol( s );
```

方法： 函数

变量： const char _far *s; 转换字符串的指针

返回值： 返回将字符串转换为一个 long 类型整数所产生的值。

B

bcopy

存储器处理函数

功能： 从一个存储区复制字符到另一个存储区。

格式：
```
#include <string.h>

void bcopy( src, dtop, size );
```

方法： 函数

变量：

char _far *src;	所要复制的存储区起始地址
char _far *dtop;	要复制到的存储区结束地址
unsigned long size;	所要复制的字节数

返回值： 不返回任何值。

功能： 从 src 指定的存储区开始将 size 指定的字节数复制到 dtop 指定的存储区。

bsearch

整数算术函数

功能： 对一个数组执行二进制搜索。

格式：
```
#include <stdlib.h>

void _far *bsearch( key, base, nelem, size, cmp );
```

方法： 函数

变量：

const void _far *key;	搜索 key
const void _far *base;	数组的起始地址
size_t nelem;	元素编号
size_t size;	元素大小
int cmp();	比较函数

返回值：
- 返回与搜索 key 相等的数组元素的指针。
- 若未找到任何匹配的元素，则将返回 NULL 的指针。

注意： 所指定的项目会在对数组进行升序排序后进行搜索。

bzero

存储器处理函数

功能： 初始化一个存储区 （将它清零）。

格式： #include <string.h>

void bzero(top, size);

方法： 函数

变量：

char _far *top;	要清零的存储区起始地址
unsigned long size;	要清零的字节数

返回值： 不返回任何值。

描述： 从 top 指定的存储区起始地址对 size 指定的字节数进行初始化 （至 0）。

C

calloc

存储器管理函数

功能： 分配一个存储区，并将它初始化为零 （0）。

格式：
```
#include <stdlib.h>

void _far * calloc( n, size );
```

方法： 函数

变量：

size_t n;	元素数
size_t size;	表示元素字节大小的值

返回值： 若找不到所指定大小的存储区，则将返回 NULL。

描述：
- 在找到指定的存储区后，它将被清零。
- 存储区的大小是两个参数的积。

规则： 存储器的保留规则与 malloc 相同。

ceil

数学函数

功能： 计算整数进位值。

格式：
```
#include <math.h>

double ceil( x );
```

方法： 函数

参数：

double x;	任意实数

返回值： 从大于指定实数 x 的整数之中返回最小的整数值。

clearerr

I/O 函数

功能： 初始化 （清除）错误状态说明符。

格式： #include <stdio.h>

void clearerr(stream);

方法： 函数

参数： FILE _far *stream; 流的指针

返回值： 不返回任何值。

描述： 将错误标志符和文件末尾标志符复位至正常值。

cos

数学函数

功能： 计算余弦值。

格式： #include <math.h>

double cos(x);

方法： 函数

参数： double x; 任意实数

返回值： 返回特定实数 x 以弧度为单位的余弦值。

cosh

数学函数

功能： 计算双曲余弦值。

格式： #include <math.h>

double cosh(x);

方法： 函数

参数： double x; 任意实数

返回值： 返回特定实数 x 的双曲余弦值。

D

div

整数算术函数

功能： 对一个 int 类型整数执行除法，并计算余数。

格式：
```
#include <stdlib.h>

div_t div( number, denom );
```

方法： 函数

参数：

int number;	被除数
int denom;	除数

返回值： 返回将 “number” 除以 “denom” 所得的商数及除法的余数。

描述：
- 将 “number” 除以 “denom” 所得的商数及除法的余数以 div_t 的结构返回。
- div_t 在 stdlib.h 中定义。组成这个结构的成员是 int quot 和 int rem。

E

exp

数学函数

功能： 计算指数函数。

格式：
```
#include <math.h>

double exp( x );
```

方法： 函数

参数：

double x;	任意实数

返回值： 返回特定实数 x 的指数函数的计算结果。

F

fabs

数学函数

功能： 计算双精度浮点数的绝对值。

格式：
```
#include <math.h>
double fabs( x );
```

方法： 函数

参数：

double x;	任意实数

返回值： 返回双精度浮点数的绝对值。

feof

I/O 函数

功能： 确认 EOF （文件末尾）。

格式：
```
#include <stdio.h>
int feof( stream );
```

方法： 宏

参数：

FILE _far *stream;	流的指针

返回值：
- 若流是 EOF，则返回 “true”（除 0 以外）。
- 否则，则返回 NULL （0）。

描述：
- 确定流的读取是否已到达 EOF。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

ferror

I/O 函数

功能： 确认输入 / 输出错误。

格式：
```
#include <stdio.h>

int ferror( stream );
```

方法： 宏

参数： FILE _far *stream; 流的指针

返回值：
- 若流存在错误，则返回 “true”（除 0 以外）。
- 否则，则返回 NULL （0）。

描述：
- 确定流中的错误。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

fflush

I/O 函数

功能： 转储清除输出缓冲器的流。

格式：
```
#include <stdio.h>

int fflush( stream );
```

方法： 函数

参数： FILE _far *stream; 流的指针

返回值： 总是返回 0。

fgetc

I/O 函数

功能： 从流读取一个字符。

格式：
```
#include <stdio.h>

int fgetc( stream );
```

方法： 函数

参数：

FILE _far *stream;	流的指针

返回值：
- 返回一个输入字符。
- 若发生错误或到达流末尾，则返回 EOF。

描述：
- 从流读取一个字符。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

fgets

I/O 函数

功能： 从流读取一个行。

格式：
```
#include <stdio.h>

char _far * fgets( buffer, n, stream );
```

方法： 函数

参数：

char _far *buffer;	存储位置的指针
int n;	最大字符数
FILE _far *stream;	流的指针

返回值：
- 若输入正常，将返回存储位置的指针 （参数所指定的相同指针）。
- 若发生错误或到达流末尾，则返回 NULL 指针。

描述：
- 读取所指定流的字符串，并将它存储在缓冲器中。
- 在输入下列任何一项之后，输入将结束：
  1. 新行字符 （‘\n’）
  2. n-1 字符
  3. 流末尾
- 空字符 （‘\0’）将被附加到输入字符串末尾。
- 新行字符 （‘\n’）将按原样存储。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

floor

数学函数

功能： 计算整数借位值。

格式： #include <math.h>

double floor(x);

方法： 函数

参数： double x; 任意实数

返回值： 实数值将被截尾以变成整数，并作为 double 类型返回。

fmod

数学函数

功能： 计算余数。

格式： #include <math.h>

double fmod(x ,y);

方法： 函数

参数： double x; 被除数
double y; 除数

返回值： 返回将被除数 x 除以除数 y 所得的余数。

fprintf

I/O 函数

功能： 输出带格式的字符到流。

格式：
```
#include <stdio.h>

int fprintf( stream, format, argument... );
```

方法： 函数

参数：

FILE _far *stream;	流的指针
const char _far *format;	字符串的指定格式的指针

返回值：
- 返回输出的字符数。
- 若发生硬件错误，则返回 EOF。

描述：
- 参数将根据格式转换为字符串，并输出到流。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。
- 格式的指定方式与 printf 相同。

fputc

I/O 函数

功能： 输出一个字符到流。

格式：
```
#include <stdio.h>

int fputc( c, stream );
```

方法： 函数

参数：

int c;	所要输出的字符
FILE _far *stream;	流的指针

返回值：
- 若输出正常，将返回输出字符。
- 若发生错误，则返回 EOF。

描述： 输出一个字符到流。

fputs

I/O 函数

功能： 输出一个行到流。

格式：
```
#include <stdio.h>

int fputs ( str, stream );
```

方法： 函数

参数：

const char _far *str;	所要输出的字符串的指针
FILE _far *stream;	流的指针

返回值：
- 若输出正常，则返回 0。
- 若发生错误，则返回 0 以外的任何值 (EOF) 。

描述： 输出一个行到流。

fread

I/O 函数

功能： 从流读取固定长度的数据

格式：
```
#include <stdio.h>

size_t fread( buffer, size, count, stream );
```

方法： 函数

参数：

void _far *buffer;	存储位置的指针
size_t size;	一个数据项目中的字节数
size_t count;	最大的数据项目数
FILE _far *stream;	流的指针

返回值： 返回已输入的数据项目数。

描述：
- 从流读取 size 指定的大小数据，并将它存储到缓冲器。这将按照 count 中指定的次数重复执行。
- 若在输入 count 中指定的数据前到达流末尾，这项函数将返回直到流末尾为止的数据项目读取数。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

free

存储器管理函数

功能： 释放所分配的存储区。

格式：
```
#include <stdlib.h>
void free( cp );
```

方法： 函数

参数：

void _far *cp;	所要释放存储区的指针

返回值： 不返回任何值。

描述：
- 释放之前使用 malloc 或 calloc 分配的存储区。
- 若您在参数中指定 NULL，则不会执行任何处理。

frexp

数学函数

功能： 将浮点数分为尾数部分和指数部分。

格式：
```
#include <math.h>
double frexp( x,prexp );
```

方法： 函数

参数：

double x;	浮点数
int _far *prexp;	存储底数为 2 的指数存储区的指针

返回值： 返回浮点数 x 尾数部分。

fscanf

I/O 函数

功能： 从流读取带格式的字符。

格式：

```
#include <stdio.h>

int fscanf( stream, format, argument... );
```

方法： 函数

参数：

FILE _far *stream;	流的指针
const char _far *format;	输入字符串的指针

返回值：

- 返回每个参数中所存储的数据项目数。
- 若从流输入了作为数据的 EOF，则将返回 EOF。

描述：

- 按照格式的指定转换从流输入的字符，并将它们存储在参数所示的变量中。
- 参数必须是对各个变量的指针。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。
- 格式的指定方式与 scanf 相同。

fwrite

I/O 函数

功能： 输出特定的数据项目到流。

格式：

```
#include <stdio.h>

size_t fwrite( buffer, size, count, stream );
```

方法： 函数

参数：

const void _far *buffer;	输出数据的指针
size_t size;	一个数据项目中的字节数
size_t count;	最大的数据项目数
FILE _far *stream;	流的指针

返回值： 返回已输出的数据项目数。

描述：

- 将具有 size 中所指定大小的数据输出到流。数据将按照 count 中指定的次数输出。
- 若在输入 count 中指定的数据量之前发生错误，这项函数将返回到该处为止已输出的数据项目数。

G

getc

I/O 函数

功能： 从流读取一个字符。

格式：
```
#include <stdio.h>
int getc( stream );
```

方法： 宏

参数： FILE _far *stream; 流的指针

返回值：
- 返回一个输入字符。
- 若发生错误或到达流末尾，则返回 EOF。

描述：
- 从流读取一个字符。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

getchar

I/O 函数

功能： 从 stdin 读取一个字符。

格式：
```
#include <stdio.h>
int getchar( void );
```

方法： 宏

参数： 不使用任何参数。

返回值：
- 返回一个输入字符。
- 若发生错误或到达文件末尾，则返回 EOF。

描述：
- 从流 (stdin) 读取一个字符。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

gets

I/O 函数

功能： 从 stdin 读取一个行。

格式：

```
#include <stdio.h>

char _far * gets( buffer );
```

方法： 函数

参数：

char _far *buffer;	存储位置的指针

返回值：

- 若输入正常，将返回存储位置的指针 （参数所指定的相同指针）。
- 若发生错误或到达文件末尾，则返回 NULL 指针。

描述：

- 读取 stdin 的字符串，并将它存储在缓冲器中。
- 行末尾的新行字符 （‘\n’）将被替换为空字符 （‘\0’）。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

I

init

I/O 函数

功能： 初始化流。

格式：
```
#include <stdio.h>

void init( void );
```

方法： 函数

参数： 不使用任何参数。

返回值： 不返回任何值。

描述：
- 初始化流。同时也调用函数中的 speed 和 init_prn，以进行 UART 和 Centronics 输出器件的初始设置。
- 一般上， init 会从启动程序进行调用。

isalnum

字符处理函数

功能： 确认字符是字母，还是数字 （A 到 Z、 a 到 z、 0 到 9）。

格式：
```
#include <ctype.h>

int isalnum( c );
```

方法： 宏

参数： int c; 要确认的字符

返回值：
- 若是字母或数字，则返回 0 以外的任何值。
- 若不是字母也不是数字，则返回 0。

描述： 确定参数中的字符的类型。

isalpha

字符处理函数

功能： 确认字符是否是字母 （A 到 Z、 a 到 z）。

格式：
```
#include <ctype.h>

int isalpha( c );
```

方法： 宏

参数：

int c;		要确认的字符

返回值：
- 若是字母，则返回 0 以外的任何值。
- 若不是字母，则返回 0。

描述： 确定参数中的字符的类型。

iscntrl

字符处理函数

功能： 确认字符是否是控制字符 （0x00 到 0x1f,0x7f）。

格式：
```
#include <ctype.h>

int iscntrl( c );
```

方法： 宏

参数：

int c;		要确认的字符

返回值：
- 若是数字，则返回 0 以外的任何值。
- 若不是控制字符，则返回 0。

描述： 确定参数中的字符的类型。

isdigit

字符处理函数

功能： 确认字符是否是数字 （0 到 9）。

格式：
```
#include <ctype.h>

int isdigit( c );
```

方法： 宏

参数：

int c;		要确认的字符

返回值：
- 若是数字，则返回 0 以外的任何值。
- 若不是数字，则返回 0。

描述： 确定参数中的字符的类型。

isgraph

字符处理函数

功能： 确认字符是否可打印 （除了空白符）（0x21 到 0x7e）。

格式：
```
#include <ctype.h>

int isgraph( c );
```

方法： 宏

参数：

int c;		要确认的字符

返回值：
- 若可打印，则返回 0 以外的任何值。
- 若不可打印，则返回 0。

描述： 确定参数中的字符的类型。

islower

字符处理函数

功能： 确认字符是否是小写字母 （a 到 z）。

格式： #include <ctype.h>

int islower(c);

方法： 宏

参数： int c; 要确认的字符

返回值：
- 若是小写字母，则返回 0 以外的任何值。
- 若不是小写字母，则返回 0。

描述： 确定参数中的字符的类型。

isprint

字符处理函数

功能： 确认字符是否可打印 （包括空白符）（0x20 到 0x7e）。

格式： #include <ctype.h>

int isprint(c);

方法： 宏

参数： int c; 要确认的字符

返回值：
- 若可打印，则返回 0 以外的任何值。
- 若不可打印，则返回 0。

描述： 确定参数中的字符的类型。

ispunct

字符处理函数

功能： 确认字符是否是标点字符。

格式：
```
#include <ctype.h>
int ispunct( c );
```

方法： 宏

参数：

int c;		要确认的字符

返回值：
- 若是标点字符，则返回 0 以外的任何值。
- 若不是标点字符，则返回 0。

描述： 确定参数中的字符的类型。

isspace

字符处理函数

功能： 确认字符是空白符、制表符，还是新行。

格式：
```
#include <ctype.h>
int isspace( c );
```

方法： 宏

参数：

int c;		要确认的字符

返回值：
- 若是空白符、制表符或新行，则返回 0 以外的任何值。
- 若不是空白符、制表符或新行，则返回 0。

描述： 确定参数中的字符的类型。

isupper

字符处理函数

功能： 确认字符是否是大写字母 （A 到 Z）。

格式：
```
#include <ctype.h>
int isupper( c );
```

方法： 宏

参数：

int c;		要确认的字符

返回值：
- 若是大写字母，则返回 0 以外的任何值。
- 若不是大写字母，则返回 0。

描述： 确定参数中的字符的类型。

isxdigit

字符处理函数

功能： 确认字符是否是十六进制字符 （0 到 9、 A 到 F、 a 到 f）。

格式：
```
#include <ctype.h>
int isxdigit( c );
```

方法： 宏

参数：

int c;		要确认的字符

返回值：
- 若是十六进制字符，则返回 0 以外的任何值。
- 若不是十六进制字符，则返回 0。

描述： 确定参数中的字符的类型。

L

labs

整数算术函数

功能： 计算 long 类型整数的绝对值。

格式：
```
#include <stdlib.h>
long labs( n );
```

方法： 函数

参数：

long n;	long 类型整数

返回值： 返回 long 类型整数的绝对值 （和 0 的距离）。

ldexp

地域化函数

功能： 计算浮点数的幂。

格式：
```
#include <math.h>
double ldexp( x,exp );
```

方法： 函数

参数：

double x;	浮点数
int exp;	数值的幂

返回值： 返回 x * （exp 2 的幂）。

ldiv

整数算术函数

功能： 对一个 long 类型整数执行除法，并计算余数。

格式：
```
#include <stdlib.h>
ldiv_t ldiv( number, denom );
```

方法： 函数

参数：

long number;	被除数
long denom;	除数

返回值： 返回将 “number”除以 “denom”所得的商数及除法的余数。

描述：
- 将 “number”除以 “denom”所得的商数及除法的余数以 ldiv_t 的结构返回。
- ldiv_t 在 stdlib.h 中定义。组成这个结构的成员是 long quot 和 long rem。

localeconv

地域化函数

功能： 初始化 struct lconv。

格式：
```
#include <locale.h>
struct lconv _far *localeconv( void );
```

方法： 函数

参数： 不使用任何参数。

返回值： 返回指向初始化 struct lconv 的指针。

log

数学函数

功能： 计算自然对数。

格式：
```
#include <math.h>
double log( x );
```

方法： 函数

参数：

double x;	任意实数

返回值： 返回特定实数 x 的自然对数。

描述： 这是 exp 的逆函数。

log10

数学函数

功能： 计算常用对数。

格式：
```
#include <math.h>
double log10( x );
```

方法： 函数

参数：

double x;	任意实数

返回值： 返回特定实数 x 的常用对数。

longjmp

指定控制函数

功能： 在进行函数调用时恢复环境

格式：
```
#include <setjmp.h>
void longjmp( env, val );
```

方法： 函数

参数：

jmp_buf env;	恢复环境的存储区的指针
int val;	值将作为 setjmp 的结果返回

返回值： 不返回任何值。

描述：
- 从以 “env” 表示的存储区恢复环境。
- 程序主控权将传递给 setjmp 调用处之后的语句。
- “val” 中指定的值将作为 setjmp 的结果返回。不过，若 “val” 是 “0”，它将被转换为 “1”。

M

malloc

存储器管理函数

功能： 分配存储区。

格式： #include <stdlib.h>

void _far * malloc(nbytes);

方法： 函数

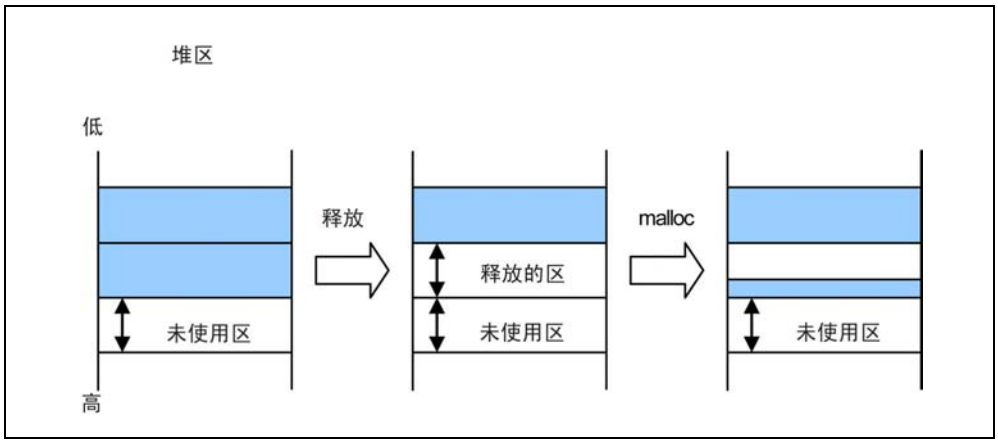
参数： size_t nbytes; 要分配的存储区大小 （按字节计算）

返回值： 若找不到所指定大小的存储区，则将返回 NULL。

描述： 动态分配存储区

规则： malloc 执行下列两项确认，以在适当的位置保留存储器。

1. 若使用 free 释放了存储区
   - 若所要保留的存储器小于所释放的存储器，则将从 free 所释放连续空区的高位地址，向低位地址保留存储区。

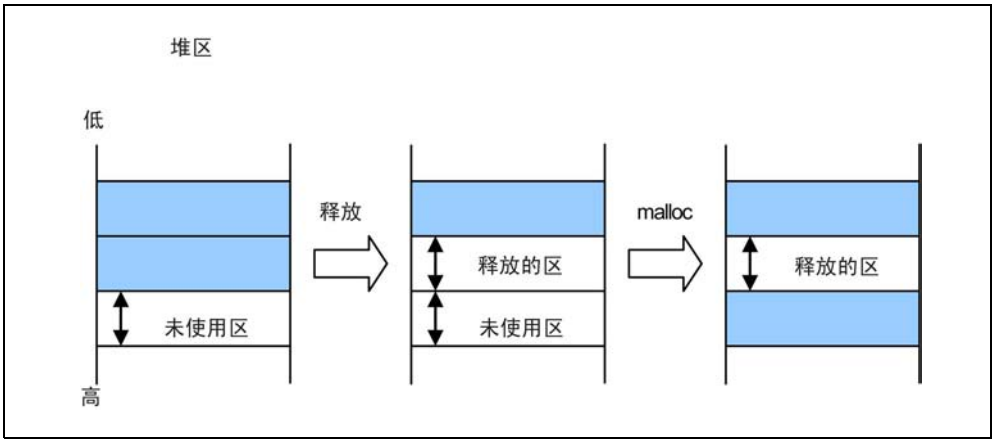


malloc

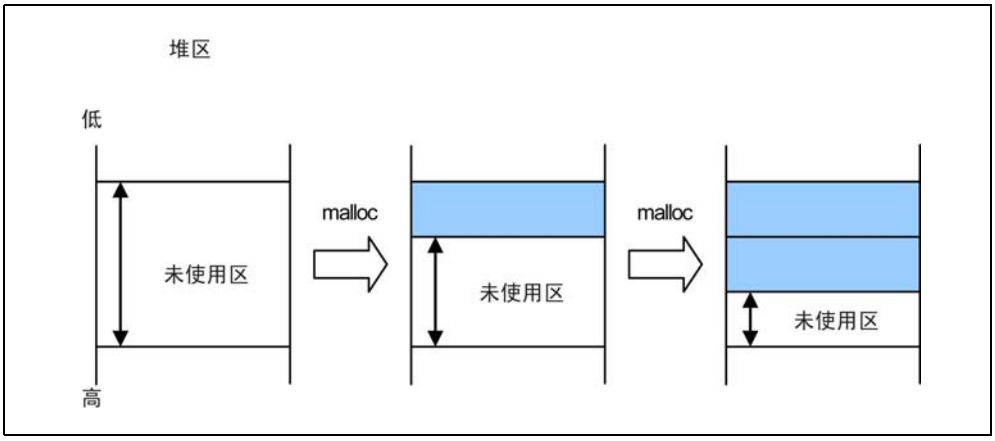
存储器管理函数

规则：

- 若所要保留的存储器大于所释放的存储器，则将从未使用的存储器的最低位地址，向高位地址保留存储区。



2. 若未使用 free 释放任何存储区
   - 若有任何可以保留的未使用存储区，则将从未使用的存储器的最低位地址，向高位地址保留存储区。



   - 若没有任何未使用的存储区可以保留，malloc 将返回 NULL，而不保留任何存储器。

注意：不会执行任何的垃圾回收。因此，即使有许多未使用的小量存储器，也没有任何存储器会被保留，同时 malloc 将返回 NULL，除非有大于所指定大小的未使用存储器。

mblen

多字节字符和多字节字符串操作函数

功能： 计算多字节字符串的长度。

格式：

```
#include <stdlib.h>

int mblen ( s,n );
```

方法： 函数

参数：

const char _far *s;	多字节字符串的指针
size_t n;	搜索的字节数

返回值：

- 若 ‘s’ 正确配置多字节字符串，则将返回字符串中的字节数。
- 若 ‘s’ 未正确配置多字节字符串，则将返回 -1。

描述：

- 若 ‘s’ 表示 NULL 字符，则将返回 0。

mbstowcs

多字节字符和多字节字符串操作函数

功能： 将多字节字符串转换为一个宽字符串。

格式：

```
#include <stdlib.h>

size_t mbstowcs( wcs,s,n );
```

方法： 函数

参数：

wchar_t _far *wcs;	存储转换宽字符串的存储区的指针
const char _far *s;	多字节字符串的指针
size_t n;	存储的宽字符数

返回值：

- 返回所转换的多字节字符串中的字符数。
- 若 ‘s’ 未正确配置多字节字符串，则将返回 -1。

mbtowc

多字节字符和多字节字符串操作函数

功能： 将多字节字符转换为一个宽字符。

格式：

```
#include <stdlib.h>

int mbtowc( wcs,s,n );
```

方法： 函数

参数：

wchar_t _far *wcs;	存储转换宽字符串的存储区的指针
const char _far *s;	多字节字符串的指针
size_t n;	存储的宽字符数

返回值：

- 若 ‘s’ 正确配置多字节字符串，则将返回所转换的宽字符数。
- 若 ‘s’ 未正确配置多字节字符串，则将返回 -1。
- 若 ‘s’ 表示 NULL 字符，则将返回 0。

memchr

存储器处理函数

功能： 从存储区搜索一个字符。

格式：

```
#include <string.h>

void _far * memchr( s, c, n );
```

方法： 函数

参数：

const void _far *s;	所要搜索的存储区的指针
int c;	所要搜索的字符
size_t n;	所要搜索的存储区的大小

返回值：

- 返回指定字符 “c” 被找到的位置 （指针）。
- 若未在存储区中找到字符 “c”，则将返回 NULL。

描述：

- 在 “n” 指定的存储器中，从 “s” 指定的地址开始，搜索在 “c” 中显示的字符。
- 当您指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

memcmp

存储器处理函数

功能： 比较存储区 （‘n’ 字节）。

格式：

```
#include <string.h>

int memcmp( s1, s2, n );
```

方法： 函数

参数：

const void _far *s1;	所要比较的第一个存储区的指针
const void _far *s2;	所要比较的第二个存储区的指针
size_t n;	所要比较的字节数

返回值：

- 返回值 = =0 两个存储区相等。
- 返回值 >0 第一个存储区大于 (s1) 另一个存储区。
- 返回值 <0 第二个存储区 (s2) 大于另一个存储区。

描述：

- 比较两个存储区的每个 n 字节。
- 当您指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

memcpy

存储器处理函数

功能： 复制存储器的 n 字节

格式：

```
#include <string.h>

void _far * memcpy( s1, s2, n );
```

方法： 宏 （默认）或函数

参数：

void _far *s1;	所要复制到的存储区的指针
const void _far *s2;	所要复制的存储区的指针
size_t n;	所要复制的字节数

返回值： 返回字符复制到的存储区的指针。

描述：

- 一般上，这项函数会使用宏所描述的程序代码。通过在一个程序库中使用函数，请在 #include <string.h> 的描述后，将它描述为 #undef memcpy。
- 从存储器 “S2” 复制 “n” 字节到存储器 “S1”。
- 当您指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

memicmp

存储器处理函数

功能： 比较存储区 （将字母作为大写字母处理）。

格式：

```
#include <string.h>

int memicmp( s1, s2, n);
```

方法： 函数

参数：

char _far *s1;	所要比较的第一个存储区的指针
char _far *s2;	所要比较的第二个存储区的指针
size_t n;	所要比较的字节数

返回值：

- 返回值 = =0 两个存储区相等。
- 返回值 >0 第一个存储区大于 (s1) 另一个存储区。
- 返回值 <0 第二个存储区 (s2) 大于另一个存储区。

描述：

- 比较存储区 （将字母作为大写字母处理）。
- 当您指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

memmove

存储器处理函数

功能： 移动字符串的存储区。

格式：

```
#include <string.h>

void _far * memmove( s1, s2, n );
```

方法： 函数

参数：

void *s1;	所要移动到的指针
const void *s2;	所要移动的指针
size_t n;	所要移动的字节数

返回值： 返回移动目的地的指针。

描述： 当您指定选项 -O[3 到 5]、 -OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

memset

存储器处理函数

功能： 设置一个存储区。

格式：
```
#include <string.h>
void _far * memset( s, c, n );
```

方法： 宏或函数

参数：

void _far *s;	所要设置的存储区的指针
int c;	所要设置的数据
size_t n;	所要设置的字节数

返回值： 返回所设置的存储区的指针。

描述：

- 一般上，这项函数会使用宏所描述的程序代码。通过在一个程序库中使用函数，请在 #include <string.h> 的描述后，将它描述为 #undef memset。
- 在存储器 “s” 中设置数据 “c” 的 “n” 字节。
- 当您指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

modf

数学函数

功能： 将实数分为尾数部分和指数部分的计算。

格式：
```
#include <math.h>
double modf ( val,pd );
```

方法： 函数

参数：

double val;	任意实数
double *pd;	存储整数的存储区的指针

返回值： 返回实数的十进制部分。

P

perror

I/O 函数

功能： 输出错误信息到 stderr。

格式：
```
#include <stdio.h>
void perror( s );
```

方法： 函数

参数：

const char _far *s;	附于信息前的字符串的指针。

返回值： 不返回任何值。

pow

数学函数

功能： 计算数值的幂。

格式：
```
#include <math.h>
double pow( x,y );
```

方法： 函数

参数：

double x;	被乘数
double y;	数值的幂

返回值： 返回被乘数 x 的 y 次幂。

printf

I/O 函数

功能： 输出带格式的字符到 stdout。

格式：
```
#include <stdio.h>

int printf( format, argument... );
```

方法： 函数

参数： const char _far *format; 字符串的指定格式的指针

格式中字符串的百分比 （%）符号之后的部分具有以下意义。 [和] 之间的部分是可选的指定。格式的详情如下所示。

格式：	%[标志][最小字段宽度][精度][修饰符 (l、 L 或 h)] 转换指定字符
范例格式：	%-05.8ld

返回值：
- 返回输出的字符数。
- 若发生硬件错误，则返回 EOF。

描述：
- 按格式中的指定将参数转换为字符串，并将字符串输出到 stdout。
- 当为参数指定指针时，必须为其指定 far 类型的指针。

1. 转换的指定符号
   - d、 I
     将参数中的整数转换为一个带符号的十进制。
   - u
     将参数中的整数转换为一个无符号的十进制。
   - o
     将参数中的整数转换为一个无符号的八进制。
   - x
     将参数中的整数转换为一个无符号的十六进制。小写的“abcdef”相等于 0AH 到 0FH。
   - X
     将参数中的整数转换为一个无符号的十六进制。大写的“ABCDEF”相等于 0AH 到 0FH。
   - c
     将参数输出为 ASCII 字符。
   - s
     将字符串 far 指针之后 (char *) 的参数 （到空字符 ‘/0’或精度为止）转换为字符串。wchar_t 类型的字符串将无法被处理。[1]
   - p
     在格式 24 位的地址中输出参数指针 （所有类型）。
   - n
     存储参数的整数指针中所输出的字符数。参数将不会被转换。

[1] 在产品随附的标准程序库中，字符串指针是 far 指针。（所有 printf 函数都将 %s 处理为 far 指针。）scanf 函数在默认情况下使用 near 指针。

printf

I/O 函数

描述：

- e

  将 double 类型的参数转换为指数格式。格式是 [-]d.dddddde±dd。
- E

  同 e，除了 E 代替 e 被用于指数。
- f

  将 double 参数转换为 [-]d.dddddd 的格式。
- g

  将 double 参数转换为 e 或 f 中指定的格式。通常使用 f 转换，但当指数为 -4 或更小，或当精度小于指数的值时，则使用 e 类型转换。
- G

  同 g，除了 E 代替 e 被用于指数。
- –

  以最小字段宽度左对齐转换结果。默认设置是右对齐。
- +

  添加 + 或 – 到带符号的转换结果。默认情况下，只有 - 被添加到负数。
- 空格 ‘ ’

  默认情况下，若带符号转换的结果没有符号，则会在值之前添加一个空格。
- #

  添加 0 到 o 转换的开头。

  当 x 或 X 转换的转换结果是 0 以外的值时，则添加 0x 或 0X 到开头。

  总是在 e、 E 和 f 转换中添加小数点。

  总是在 g 和 G 转换中添加小数点，同时输出小数位中的任何 0。

2. 最小字段宽度
   - 指定十进制正整数的最小字段宽度。
   - 当转换结果具有的字符比指定的字段宽度少时，字段的左侧将被填充。
   - 默认的填充字符是空格。不过，若使用前面有 ‘0’ 的整数来指定字段，则 ‘0’ 将是填充字符。
   - 若您指定了 – 标志，转换结果将进行左对齐，同时填充字符 （总是为空格）将插入右侧。
   - 若您为最小字段宽度指定了星号 （*），字段宽度将由参数中的整数指定。若参数的值是负数，标志之后的值将是正数字段宽度。

3. 精度

   在 ‘.’ 之后指定一个正整数。若您指定了没有任何值的 ‘.’，它将被解释为零。函数与默认值因转换类型而异。

   浮点类型数据的输出在默认情况下具有 6 的精度。不过，若指定了 0 的精度，则不会输出任何小数位。
   - d、 i、 o、 u、 x 和 X 转换
     1. 若转换结果中的列数比指定的数目少，则会用零填充值的开头。
     2. 若指定的列数超出了最小字段宽度，将以指定的列数为准。

printf

I/O 函数

描述：

3. 若所指定精度中的列数少于最小字段宽度，则将在处理最小列数之后处理字段宽度。
4. 默认值是 1。
5. 若最小列数是零的转换值为零，则没有任何输出。

- s 转换
  1. 表达最大字符数。
  2. 若转换结果超出了指定的字符数，则将丢弃余数。
  3. 默认情况下，字符数是没有限制的。
  4. 若您为精度指定了星号 （*），则精度将由参数的整数指定。
  5. 若参数是负数值，精度的指定将无效。
- e、 E 和 f 转换
  n （n 是精度）数值在小数点后输出。
- g 和 G 转换
  超出 n （n 是精度）的字符将不会被输出。

4. I、 L 或 h
   - I：d、 i、 o、 u、 x、 X 和 n 转换在 long int 和 unsigned long int 参数上执行。
   - h：d、 i、 o、 u、 x 和 X 转换在 short int 和 unsigned short int 参数上执行。
   - 若在 d、 i、 o、 u、 x、 X 或 n 以外的转换中指定了 I 或 h，它们将被忽略。
   - L：e、 E、 f、 g 和 G 转换在 double 参数上执行。[2]

[2] 在标准的 C 指定中，变量 e、 E、 f 和 g 在 L 中的转换将在 long double 参数上执行。在 NC30 中， long double 类型被处理为 double 类型。因此，若您指定了 L，参数将作为 double 类型处理。

putc

I/O 函数

功能： 输出一个字符到流。

格式：
```
#include <stdio.h>
int putc( c, stream );
```

方法： 宏

参数：

int c;	所要输出的字符
FILE _far *stream;	流的指针

返回值：
- 若输出正常，将返回输出字符。
- 若发生错误，则返回 EOF。

描述： 输出一个字符到流。

putchar

I/O 函数

功能： 输出一个字符到 stdout。

格式：
```
#include <stdio.h>
int putchar( c );
```

方法： 宏

参数：

int c;	所要输出的字符

返回值：
- 若输出正常，将返回输出字符。
- 若发生错误，则返回 EOF。

描述： 输出一个字符到 stdout。

puts

I/O 函数

功能： 输出一个行到 stdout。

格式：
```
#include <stdio.h>

int puts( str );
```

方法： 宏

参数：

char _far *str;		所要输出的字符串的指针

返回值：
- 若输出正常，则返回 0。
- 若发生错误，则返回 -1 (EOF) 。

描述：
- 输出一个行到 stdout。
- 字符串末尾的空字符 （‘\0’）将被替换为新行字符 （‘\n’）。

Q

qsort

整数算术函数

功能： 为数组中的元素进行排序。

格式：
```
#include <stdlib.h>

void qsort( base,nelen,size,cmp( e1,e2 ) );
```

方法： 函数

参数：

void _far *base;	数组的起始地址
size_t nelen;	元素编号
size_t size;	元素大小
int cmp();	比较函数

返回值： 不返回任何值。

描述： 为数组中的元素进行排序。

R

rand

整数算术函数

功能： 生成一个伪随机数。

格式：
```
#include <stdlib.h>

int rand( void );
```

方法： 函数

参数： 不使用任何参数。

返回值：
- 返回在 srand 中指定的种子随机数系列。
- 所生成的随机数是介于 0 和 RAND_MAX 之间的值。

realloc

存储器管理函数

功能： 更改所分配的一个存储区的大小。

格式：
```
#include <stdlib.h>

void _far * realloc( cp, nbytes );
```

方法： 函数

参数：

void _far *cp;	更改前的存储区的指针
size_t nbytes;	要更改的存储区大小 （按字节计算）

返回值：
- 返回更改了大小的存储区的指针。
- 若无法保留指定大小的存储区，则将返回 NULL。

描述：
- 更改已使用 malloc 或 calloc 进行保留的存储区的大小。
- 在参数 “cp”中指定一个之前已保留的指针，并在 “nbytes”中指定所要更改的字节数。

S

scanf

I/O 函数

功能： 从 stdin 读取带格式的字符。

格式：

```
#include <stdio.h>
#include <ctype.h>

int scanf( format, argument... );
```

方法： 函数

参数： const char _far *format; 字符串的指定格式的指针

格式中字符串的百分比 （%）符号之后的部分具有下列意义 [和] 之间的部分是可选的指定。格式的详情如下所示。

格式：	%[*][最大字段宽度][修饰符 (l、 L 或 h)] 转换指定字符
范例格式：	%*5ld

返回值：

- 返回每个参数中所存储的数据项目数。
- 若从 stdin 输入了作为数据的 EOF，则将返回 EOF。

描述：

- 按照格式的指定转换从 stdin 读取的字符，并将它们存储在 argument 所示的变量中。
- Argument 必须是对各个变量的 far 指针。
- 除了在 c 和 [ ] 转换中外，第一个空格字符将被忽略。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

1. 转换的指定符号
   - d
     转换一个带符号的十进制。目标参数必须是对整数的指针。
   - i
     转换带符号的十进制、八进制和十六进制输入。八进制以 0 开头。
     十六进制以 0x 或 0X 开头。目标参数必须是对整数的指针。
   - u
     转换一个无符号的十进制。目标参数必须是对无符号整数的指针。
   - o
     转换一个带符号的八进制。目标参数必须是对整数的指针。
   - x、 X
     转换一个带符号的十六进制。大写或小写字母都可用于 0AH 到 0FH。这不包括开头的 0x。目标参数必须是对整数的指针。

scanf

I/O 函数

描述：

- s

  存储以空字符 ‘\0’结尾的字符串。目标参数必须是对具有足够大小的字符数组的指针，该数组可存储包括空字符 ‘\0’的字符串。
  若输入在到达最大字段宽度时停止，所存储的字符串将包括到停止处为止的字符，加上结尾的空字符。
- c

  存储一个字符，空格字符不会被跳过。若为最大字段宽度指定 2 或更大的值，则会有多个字符被存储。不过，这并不包括空字符 ‘\0’。目标参数必须是对具有足够大小可存储字符串的字符数组的指针。
- p

  转换格式数据存储体寄存器中的数据，加上偏移 （范例：00:1205). 目标参数是对所有类型的指针。
- [ ]

  当输入了 [和] 之间的一个或多个字符时，存储所输入的字符。当所输入的不是 [和] 之间的字符时，存储将停止。若在 [之后指定了声调符号（^），只有声调符号和] 之间以外的字符是有效的输入字符。当输入了其中一个指定的字符时，存储将停止。
  目标参数必须是对具有足够大小的字符数组的指针，该数组可存储包括自动添加的空字符 ‘\0’的字符串。
- n

  存储已在格式转换中读取的字符数。目标参数必须是对整数的指针。
- e、 E、 f、 g、 G

  转换到浮点格式。若指定了修饰符 I，目标参数必须是 double 类型的指针。而默认设置则是 float 类型的指针。

2. *（阻止数据存储）
   - 指定星号 （*）将阻止参数中所转换的数据的存储。
3. 最大字段宽度
   - 将最大输入字符数指定为十进制正整数。在任何一种格式转换中，所读取的字符数将不会超出这个数字。
   - 若在读取指定的字符数前，输入了空格字符 （在函数 isspace() 中为真的字符）或指定格式以外的字符，则读取将停在该字符上。
4. I、 L 或 h
   - I：d、i、o、u 和 x 转换的结果将存储为 long int 和 unsigned long int。e、E、f、g 和 G 转换的结果则将存储为 double。
   - h：d、 i、 o、 u 和 x 转换的结果将存储为 short int 和 unsigned short int。
   - 若在 d、 i、 o、 u 或 x 以外的转换中指定了 I 或 h，它们将被忽略。
   - L：e、 E、 f、 g 和 G 转换的结果则将存储为 float。

setjmp

指定控制函数

功能： 在函数调用前保存环境

格式： #include <setjmp.h>

int setjmp(env);

方法： 函数

参数： jmp_buf env; 保存环境的存储区的指针

返回值： 返回参数 longjmp 所给予的数值。

描述： 在 “env”指定的存储区中保存环境。

setlocale

地域化函数

功能： 设置及搜索程序的区域信息。

格式： #include <locale.h>

char _far *setlocale(category,locale);

方法： 函数

参数： int category; 区域信息、搜索段信息
const char _far *locale; 区域信息字符串的指针

返回值：
- 返回区域信息字符串的指针。
- 若无法设置或搜索信息，则返回 NULL。

sin

数学函数

功能： 计算正弦值。

格式： #include <math.h>

double sin(x);

方法： 函数

参数： double x; 任意实数

返回值： 返回特定实数 x 以弧度为单位的正弦值。

sinh

数学函数

功能： 计算双曲正弦值。

格式： #include <math.h>

double sinh(x);

方法： 函数

参数： double x; 任意实数

返回值： 返回特定实数 x 的双曲正弦值。

sprintf

I/O 函数

功能： 写入带格式的文本到字符串。

格式： #include <stdio.h>

int sprintf(pointer, format, argument...);

方法： 函数

参数： char _far *pointer; 存储位置的指针
const char _far *format; 字符串的指定格式的指针

返回值： 返回输出的字符数。

描述：
- 按照 format 的指定将 argument 转换为字符串，并将它们存储到 pointer。
- format 的指定方式与 printf 相同。

sqrt

数学函数

功能： 计算数值的平方根。

格式： #include <math.h>

double sqrt(x);

方法： 函数

参数： double x; 任意实数

返回值： 返回特定实数 x 的平方根。

srand

整数算术函数

功能： 将种子给予生成例程的伪随机数。

格式：
```
#include <stdlib.h>
void srand( seed );
```

方法： 函数

参数：

unsigned int seed;	随机数字的序列值

返回值： 不返回任何值。

描述： 使用 seed 初始化 （种子） rand 所产生的伪随机数系列。

sscanf

I/O 函数

功能： 从字符串读取带格式的数据。

格式：
```
#include <stdio.h>
int sscanf( string, format, argument... );
```

方法： 函数

参数：

const char _far *string;	输入字符串的指针
const char _far *format;	字符串的指定格式的指针

返回值：
- 返回每个参数中所存储的数据项目数。
- 若空字符 （‘\0’）作为数据输入，则返回 EOF。

描述：
- 按照 format 的指定转换输入的字符，并将它们存储在 argument 所示的变量中。
- Argument 必须是对各个变量的 far 指针。
- format 的指定方式与 scanf 相同。

strcat

字符串处理函数

功能： 连接字符串。

格式：

```
#include <string.h>

char _far * strcat( s1, s2 );
```

方法： 函数

参数：

char _far *s1;	所要连接到的字符串的指针
const char _far *s2;	所要连接的字符串的指针

返回值： 返回所连接字符串的存储区 (s1) 的指针。

描述：

- 根据 s1+s2 的顺序连接字符串 “s1” 和 “s2” [3]。
- 连接的字符串以 NULL 结尾。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strchr

字符串处理函数

功能： 从字符串顶端开始搜索特定的字符。

格式：

```
#include <string.h>

char _far * strchr( s, c );
```

方法： 函数

参数：

const char _far *s;	所要搜索的字符串的指针
int c;	所要搜索的字符

返回值：

- 返回第一次在字符串 “s” 中找到字符 “c” 的位置。
- 当字符串 “s” 不包含字符 “c” 时，则返回 NULL。

描述：

- 从存储区 “s” 的开头部分开始搜索字符 “c”。
- 也可以搜索 ‘\0’。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

3. 必须有足够的空间可容纳 s1 加 s2。

strcmp

字符串处理函数

功能： 比较字符串。

格式：
```
#include <string.h>

int strcmp( s1, s2 );
```

方法： 宏、函数

参数：

const char _far *s1;	所要比较的第一个字符串的指针
const char _far *s2;	所要比较的第二个字符串的指针

返回值：

- 返回值 = =0　两个字符串相等。
- 返回值 >0　第一个字符串大于 (s1) 另一个字符串。
- 返回值 <0　第二个字符串大于 (s2) 另一个字符串。

描述：

- 一般上，这项函数会使用宏所描述的程序代码。通过在一个程序库中使用函数，请在 #include <string.h> 的描述后，将它描述为 #undef strcmp。
- 比较两个以 NULL 结尾的字符串的每个字节。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strcoll

字符串处理函数

功能： 比较字符串 （使用区域信息）。

格式：
```
#include <string.h>

int strcoll( s1, s2 );
```

方法： 函数

参数：

const char _far *s1;	所要比较的第一个字符串的指针
const char _far *s2;	所要比较的第二个字符串的指针

返回值：

- 返回值 = =0　两个字符串相等。
- 返回值 >0　第一个字符串大于 (s1) 另一个字符串。
- 返回值 <0　第二个字符串大于 (s2) 另一个字符串。

描述： 当指定选项 -O[3 到 5]、 -OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strcpy

字符串处理函数

功能： 复制一个字符串。

格式：

```
#include <string.h>
char _far * strcpy( s1, s2 );
```

方法： 宏或函数

参数：

char _far *s1;	所要复制到的字符串的指针
const char _far *s2;	所要复制的字符串的指针

返回值： 返回复制字符串的目的地的指针。

描述：

- 一般上，这项函数会使用宏所描述的程序代码。通过在一个程序库中使用函数，请在 #include <string.h> 的描述后，将它描述为 #undef strcpy。
- 将字符串 “s2”（以 NULL 结尾）复制到存储区 “s1”。
- 复制后，字符串将具有 NULL 的结尾。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择可通过优化取得良好代码效率的函数。

strcspn

字符串处理函数

功能： 计算未在另一字符串中找到的未指定字符的长度 （字数）。

格式：

```
#include <string.h>
size_t strcspn( s1, s2 );
```

方法： 函数

参数：

const char _far *s1;	所要搜索的目标字符串的指针
const char _far *s2;	所要搜索的字符串的指针

返回值： 返回未指定字符的长度 （数值）。

描述：

- 从存储区 ‘s1’ 计算包含 ‘s2’ 以外字符的第一个字符串的大小，然后从 ‘s1’ 开始搜索字符。
- 不能搜索 ‘\0’。

stricmp

字符串处理函数

功能： 比较字符串。（所有字母作为大写字母处理。）

格式：
```
#include <string.h>

int stricmp( s1, s2 );
```

方法： 函数

参数：

char _far *s1;	所要比较的第一个字符串的指针
char _far *s2;	所要比较的第二个字符串的指针

返回值：

- 返回值 = =0　两个字符串相等。
- 返回值 >0　第一个字符串大于 (s1) 另一个字符串。
- 返回值 <0　第二个字符串大于 (s2) 另一个字符串。

描述： 比较两个以 NULL 结尾的字符串的每个字节。不过，所有字母将被处理为大写字母。

strerror

字符串处理函数

功能： 将错误编号转换为一个字符串。

格式：
```
#include <string.h>

char _far * strerror( errcode );
```

方法： 函数

参数：

int errcode;	错误代码

返回值： 返回错误代码的信息字符串的指针。

描述： stderr 返回静态数组的指针。

strlen

字符串处理函数

功能： 计算一个字符串中的字符数。

格式：
```
#include <string.h>
size_t strlen( s );
```

方法： 函数

参数：

const char _far *s;	要计算长度的字符串的指针

返回值： 返回字符串的长度。

描述： 确定字符串 “s”（到 NULL）的长度。

strncat

字符串处理函数

功能： 连接字符串 （‘n’ 字符）。

格式：
```
#include <string.h>
char _far * strncat( s1, s2, n );
```

方法： 函数

参数：

char _far *s1;	所要连接到的字符串的指针
const char _far *s2;	所要连接的字符串的指针
size_t n;	所要连接的字符数

返回值： 返回所连接字符串的存储区的指针。

描述：
- 连接字符串 “s1”和字符串 “s2”的 “n”字符。
- 连接的字符串以 NULL 结尾。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strncmp

字符串处理函数

功能： 比较字符串 （‘n’ 字符）。

格式：

```
#include <string.h>

int strncmp( s1, s2, n );
```

方法： 函数

参数：

const char _far *s1;	所要比较的第一个字符串的指针
const char _far *s2;	所要比较的第二个字符串的指针
size_t n;	所要比较的字符数

返回值：

- 返回值 = =0 两个字符串相等。
- 返回值 >0 第一个字符串大于 (s1) 另一个字符串。
- 返回值 <0 第二个字符串大于 (s2) 另一个字符串。

描述：

- 比较两个以 NULL 结尾字符串中 n 字符的每个字节。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strncpy

字符串处理函数

功能： 复制一个字符串 （‘n’ 字符）。

格式：

```
#include <string.h>

char _far * strncpy( s1, s2, n );
```

方法： 函数

参数：

char _far *s1;	所要复制到的字符串的指针
const char _far *s2;	所要复制的字符串的指针
size_t n;	所要复制的字符数

返回值： 返回复制字符串的目的地的指针。

描述：

- 从字符串 “s2” 复制 “n” 字符到存储区 “s1”。若字符串 “s2” 所包含的字符比 “n” 指定的更多，它们将不会被复制，同时也不会附加 ‘\0’。相反的，若 “s2” 所包含的字符比 “n” 指定的更少，则将附加 ‘\0’ 到所复制字符串的结尾，以凑足 “n” 所指定的数目。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strnicmp

字符串处理函数

功能： 比较字符串 （‘n’ 字符）。（所有字母作为大写字母处理。）

格式：

```
#include <string.h>

int strnicmp( s1, s2, n );
```

方法： 函数

参数：

char _far *s1;	所要比较的第一个字符串的指针
char _far *s2;	所要比较的第二个字符串的指针
size_t n;	所要比较的字符数

返回值：

- 返回值 = =0 两个字符串相等。
- 返回值 >0 第一个字符串大于 (s1) 另一个字符串。
- 返回值 <0 第二个字符串大于 (s2) 另一个字符串。

描述：

- 比较两个以 NULL 结尾字符串中 n 字符的每个字节。不过，所有字母将作为大写字母处理。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strpbrk

字符串处理函数

功能： 从另一字符串搜索字符串中的特定字符。

格式：

```
#include <string.h>

char _far * strpbrk( s1, s2 );
```

方法： 函数

参数：

const char _far *s1;	所要搜索的目标字符串的指针
const char _far *s2;	所要搜索的字符所在字符串的指针

返回值：

- 返回第一次找到指定的字符串的位置 （指针）。
- 若找不到指定的字符，则返回 NULL。

描述：

- 在 “s1” 存储区的字符串中搜索指定的字符 “s2”。
- 不能搜索 ‘\0’。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strrchr

字符串处理函数

功能： 从字符串的末尾搜索特定字符。

格式：
```
#include <string.h>

char _far * strrchr( s, c );
```

方法： 函数

参数：

const char _far *s;	所要搜索的目标字符串的指针
int c;	所要搜索的字符

返回值：
- 返回最后一次在字符串 “s”中找到字符 “c”的位置。
- 当字符串 “s”不包含字符 “c”时，则返回 NULL。

描述：
- 从存储区 “s”的末尾搜索 “c”所指定的字符。
- 可以搜索 ‘\0’。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strspn

字符串处理函数

功能： 计算在字符串中找到的指定字符的长度 （字数）。

格式：
```
#include <string.h>

size_t strspn( s1, s2 );
```

方法： 函数

参数：

const char _far *s1;	所要搜索的目标字符串的指针
const char _far *s2;	所要搜索的字符所在字符串的指针

返回值： 返回指定字符的长度 （数值）。

描述：
- 从存储区 ‘s1’计算包含 ‘s2’中字符的第一个字符串的大小，然后从 ‘s1’开始搜索字符。
- 不能搜索 ‘\0’。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strstr

字符串处理函数

功能： 从字符串搜索特定字符。

格式：

```
#include <string.h>

char _far * strstr( s1, s2 );
```

方法： 函数

参数：

const char _far *s1;	所要搜索的目标字符串的指针
const char _far *s2;	所要搜索的字符所在字符串的指针

返回值：

- 返回找到指定的字符串的位置 （指针）。
- 若找不到指定的字符，则返回 NULL。

描述：

- 返回第一个字符串 “s2”从存储区 “s1”的开头部分算起的位置 （指针）。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strtod

转换字符串的值的函数

功能： 将字符串转换为一个 double 类型整数。

格式：

```
#include <string.h>

double strtod( s,endptr );
```

方法： 函数

参数：

const char _far *s;	转换字符串的指针
char _far * _far *endptr;	未被转换的其余字符串的指针

返回值：

• 返回值 = = 0L	不构成一个数值。
• 返回值 != 0L	返回 double 类型的配置数值。

描述： 当指定选项 -O[3 到 5]、 -OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strtok

字符串处理函数

功能： 在一个字符串中将其中一些字符串划分为标志。

格式：

```
#include <string.h>

char _far * strtok( s1, s2 );
```

方法： 函数

参数：

char _far *s1;	所要划分的字符串的指针
const char _far *s2;	划分所使用的标点字符的指针

返回值：

- 返回找到字符时的划分标志的指针。
- 若找不到字符，则返回 NULL。

描述：

- 在第一次调用中，返回第一个标志的第一个字符的指针。在所返回的字符之后将写入 NULL 字符。在接下来的调用中 （当 “s1”为 NULL 时），这个指令将返回它所找到的每个标志。当 “s1”中的标志已全部找到时，则返回 NULL。
- 当指定选项 -O[3 到 5]、-OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strtol

转换字符串的值的函数

功能： 将字符串转换为一个 long 类型整数。

格式：

```
#include <string.h>

long strtol( s,endptr,base );
```

方法： 函数

参数：

const char _far *s;	转换字符串的指针
char _far * _far *endptr;	未被转换的其余字符串的指针
int base;	读取值时的底数 （0 到 36）
	若值的底数是零，则读取整数常数的格式

返回值：

• 返回值 = = 0L	不构成一个数值。
• 返回值 != 0L	返回 long 类型的配置数值。

描述： 当指定选项 -O[3 到 5]、 -OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strtoul

转换字符串的值的函数

功能： 将字符串转换为一个 unsigned long 类型整数。

格式：

```
#include <string.h>

unsigned long strtoul( s,endptr,base );
```

方法： 函数

参数：

const char _far *s;	转换字符串的指针
char _far * _far *endptr;	未被转换的其余字符串的指针
int base;	读取值时的底数 （0 到 36） 若值的底数是零，则读取整数常数的格式

返回值：

- 返回值 = = 0L　不构成一个数值。
- 返回值 != 0L　返回 long 类型的配置数值。

描述： 当指定选项 -O[3 到 5]、 -OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

strxfrm

转换字符串的值的函数

功能： 转换一个字符串 （使用区域信息）。

格式：

```
#include <string.h>

size_t strxfrm( s1,s2,n );
```

方法： 函数

参数：

char _far *s1;	存储转换结果字符串的存储区的指针
const char _far *s2;	所要转换的字符串的指针
size_t n;	所转换的字节数

返回值： 返回所转换的字符数。

描述： 当指定选项 -O[3 到 5]、 -OR 或 -OS 时，系统可能会选择其它可通过优化取得良好代码效率的函数。

T

tan

数学函数

功能： 计算正切值。

格式：
```
#include <math.h>
double tan( x );
```

方法： 函数

参数：

double x;		任意实数

返回值： 返回特定实数 x 以弧度为单位的正切值。

tanh

数学函数

功能： 计算双曲正切值。

格式：
```
#include <math.h>
double tanh( x );
```

方法： 函数

参数：

double x;		任意实数

返回值： 返回特定实数 x 的双曲正切值。

tolower

字符处理函数

功能： 将字符从大写转换为小写。

格式： #include <ctype.h>

int tolower(c);

方法： 宏

参数： int c; 所要转换的字符

返回值：
- 若参数是大写字母，则返回小写字母。
- 否则，则原样返回所传递的参数。

描述： 将字符从大写转换为小写。

toupper

字符处理函数

功能： 将字符从小写转换为大写。

格式： #include <ctype.h>

int toupper(c);

方法： 宏

参数： int c; 所要转换的字符

返回值：
- 若参数是小写字母，则返回大写字母。
- 否则，则原样返回所传递的参数。

描述： 将字符从小写转换为大写。

U

ungetc

I/O 函数

功能： 返回一个字符到流。

格式：
```
#include <stdio.h>

int ungetc( c, stream );
```

方法： 宏

参数：

int c;	要返回的字符
FILE _far *stream;	流的指针

返回值：
- 若操作正常完成，则返回所返回的一个字符。
- 若流处于写入模式、发生错误或到达 EOF，抑或所要返回的字符是 EOF，则返回 EOF。

描述：
- 返回一个字符到流。
- 将代码 0x1A 解释为结束代码，并忽略之后的任何数据。

V

vfprintf

I/O 函数

功能： 对流进行带格式的输出。

格式：
```
#include <stdarg.h>
#include <stdio.h>

int vfprintf( stream, format, ap... );
```

方法： 函数

参数：

FILE _far *stream;	流的指针
const char _far *format;	字符串的指定格式的指针
va_list ap;	参数列表的指针

返回值： 返回输出的字符数。

描述：
- 对流进行带格式的输出。
- 当在变量长度的变量中编写指针时，确保它们是 far 类型的指针。

vprintf

I/O 函数

功能： 对 stdout 进行带格式的输出。

格式：
```
#include <stdarg.h>
#include <stdio.h>

int vprintf( format, ap... );
```

方法： 函数

参数：

const char _far *format;	字符串的指定格式的指针
va_list ap;	参数列表的指针

返回值： 返回输出的字符数。

描述：
- 对 stdout 进行带格式的输出。
- 当在变量长度的变量中编写指针时，确保它们是 far 类型的指针。

vsprintf

I/O 函数

功能： 对缓冲器进行带格式的输出。

格式：
```
#include <stdarg.h>
#include <stdio.h>

int vfprintf( s, format, ap... );
```

方法： 函数

参数：

char _far *s;	存储位置的指针
const char _far *format;	字符串的指定格式的指针
va_list ap;	参数列表的指针

返回值： 返回输出的字符数。

描述： 当在变量长度的变量中编写指针时，确保它们是 far 类型的指针。

W

wcstombs

多字节字符和多字节字符串操作函数

功能： 将宽字符串转换为一个多字节字符串。

格式：
```
#include <stdlib.h>

size_t _far wcstombs( s, wcs, n );
```

方法： 函数

参数：

char _far *s;	存储转换多字节字符串的存储区的指针
const wchar_t _far *wcs;	宽字符串的指针
size_t n;	存储的宽字符数

返回值：
- 若字符串正确转换，则返回所存储的多字节字符数。
- 若字符串未正确转换，则返回 -1。

wctomb

多字节字符和多字节字符串操作函数

功能： 将宽字符转换为一个多字节字符。

格式：
```
#include <stdlib.h>]

int wctomb( s,wchar );
```

方法： 函数

参数：

char _far *s;	存储转换多字节字符串的存储区的指针
wchar_t wchar;	宽字符

返回值：
- 返回多字节字符中所包含的字节数。
- 若没有对应的多字节字符，则返回 -1。
- 若宽字符为 0，则返回 0。

附录 E.2.4 使用标准程序库

(a) 有关标准标题文件的注意事项

当在标准程序库中使用函数时，总是确保包含所指定的标准标题文件。若未包含这个标题文件，则参数和返回值将失去完整性，使程序无法正确操作。

(b) 有关标准程序库的优化的注意事项

若指定了 -O[3 到 5]、 -OS 或 -OR 中的任何优化选项，系统将为标准函数执行优化。这项优化可通过指定 -Ono_stdlib 来禁止。当所使用的用户函数名称与其中一个标准程序库函数相同时，则有必要禁止这项优化。

(1) 函数的直接插入填充

对于函数 strcpy 和 memcpy，若符合附表 E.13 中的条件，系统将执行函数的直接插入填充。

附表 E.13 标准程序库函数的优化条件

函数名称	优化条件	描述范例
strcpy	第一个参数：far 指针 第二个参数：字符串常数	strcpy(str, "sample");
memcpy	第一个参数：far 指针 第二个参数：far 指针 第三个参数：常数	memcpy(str ,"sample", 6); memcpy(str , fp, 6);

附录 E.3 修改标准程序库

NC30 产品封装包含了一个全面的函数程序库，其中所包括的函数有 scanf 和 printf I/O 函数等。这些函数通常被用于调用高级的 I/O 函数。而这些高级 I/O 函数就是硬件相关的低级 I/O 函数的组合。

在 M16C/80 系列的应用程序中， I/O 函数可能需要根据目标系统的硬件进行修改。这可通过修改标准程序库的源文件来完成。

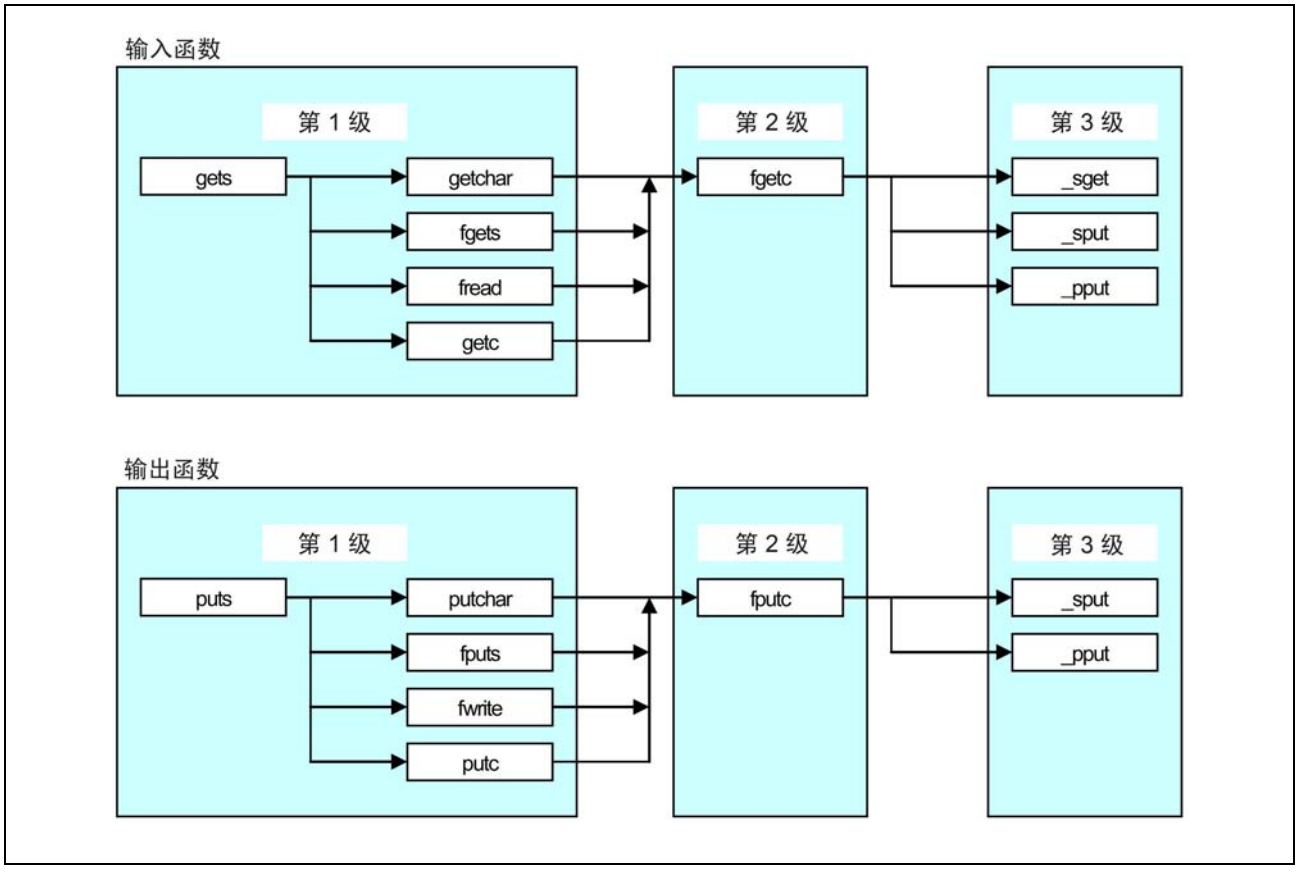
本章将说明如何修改 NC30 的标准程序库，以符合目标系统的要求。

标准函数程序库的源文件并未在评估版本中提供。因此，在评估版本中无法定制标准函数程序库。

附录 E.3.1 I/O 函数的结构

如附图 E.1 中所示， I/O 函数通过从第 1 级的函数调用等级较低的函数 （第 2 级、第 3 级）来进行操作。例如， fgets 调用第 2 级的 fgetc，而 fgetc 则调用第 3 级的函数。

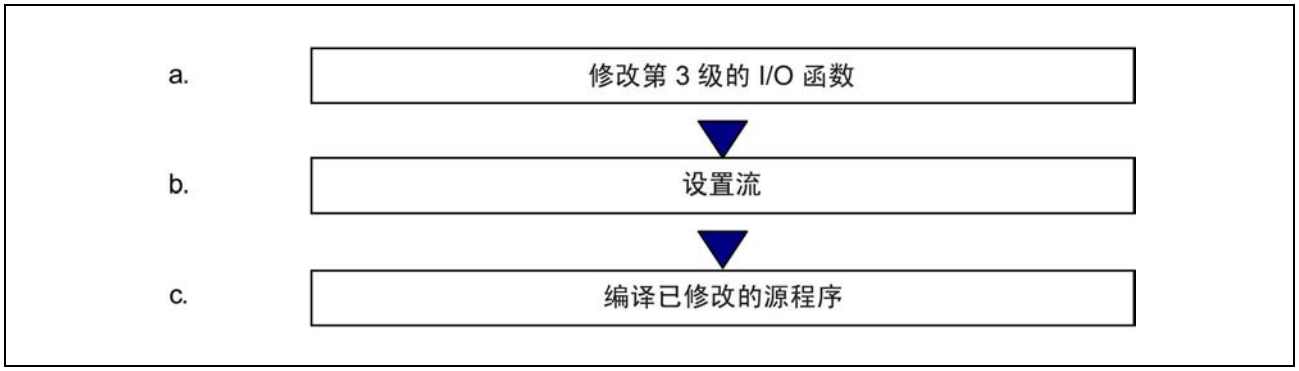
只有等级最低的第 3 级函数与单片机的硬件相关 （与 I/O 端口相关）。若您的应用程序使用 I/O 函数，您可能需要修改第 3 级函数的源文件，以符合系统的要求。



附图 E.1 I/O 函数的调用关系

附录 E.3.2 修改 I/O 函数的顺序

附图 E.2 中概述了如何修改 I/O 函数，以符合目标系统的要求。



附图 E.2 修改 I/O 函数的范例顺序

(a) 修改第 3 级的 I/O 函数

第 3 级的 I/O 函数通过 M16C/60 系列的 I/O 端口执行 1 字节的 I/O。第 3 级的 I/O 函数包括了通过串行通信电路 (UART) 执行 I/O 的 _sget 和 _sput，以及通过 Centronics 通信电路执行 I/O 的 _pput。

(1) 电路设置

- 处理器模式：单片机模式
- 时钟频率：20MHz
- 外部总线大小：16 位

(2) 初始的串行通信设置

- 使用 UART1
- 波特率：9600bps
- 数据大小：8 位
- 奇偶性：无
- 停止位：2 位

* 串行通信的初始设置在 init 函数 (init.c) 中进行。

第 3 级 的 I/O 函数在 C 程序库源文件 device.c 中编写。附表 E.14 中列出了这些函数的指定。

附表 E.14　第 3 级函数的指定

输入函数	参数	返回值 （int 类型）
_sget _sput _pput	无。	若没有发生错误，将返回输入字符。若发生错误，则将返回 EOF。

输出函数	参数 （int 类型）	返回值 （int 类型）
_sput _pput	所要输出的 字符	若没有发生错误，将返回 1。 若发生错误，则将返回 EOF。

串行通信在 M16C/80 系列的两个 UART 中设置为 UART1。可在 device.c 中编写如下，以使用条件编译命令来选择 UART0：

- 若要使用 UART0..... #define UART0 1

在 device.c 的开头部分指定这些命令，或在编译时指定下列选项。

- 若要使用 UART0..... -DUART0

若要使用两个 UART，则修改文件如下：

1. 删除 device.c 文件开头部分的条件编译命令。
2. 将 UART0 特殊寄存器在 #pragma EQU 中定义的名称更改为 UART1 以外的变量。
3. 复制第 3 级函数 _sget 和 UART0 的 _sput，并将它们更改为不同的变量名称，如 _sget0 和 _sput0。
4. 同时也复制 UART0 的 speed 函数，并将函数名称更改为 speed0 之类的名称。

这就完成了 device.c 的修改。

接着，修改进行 I/O 函数的初始设置的 init 函数 (init.c)，然后更改流的设置 （参考下面的做法）。

(b)　流的设置

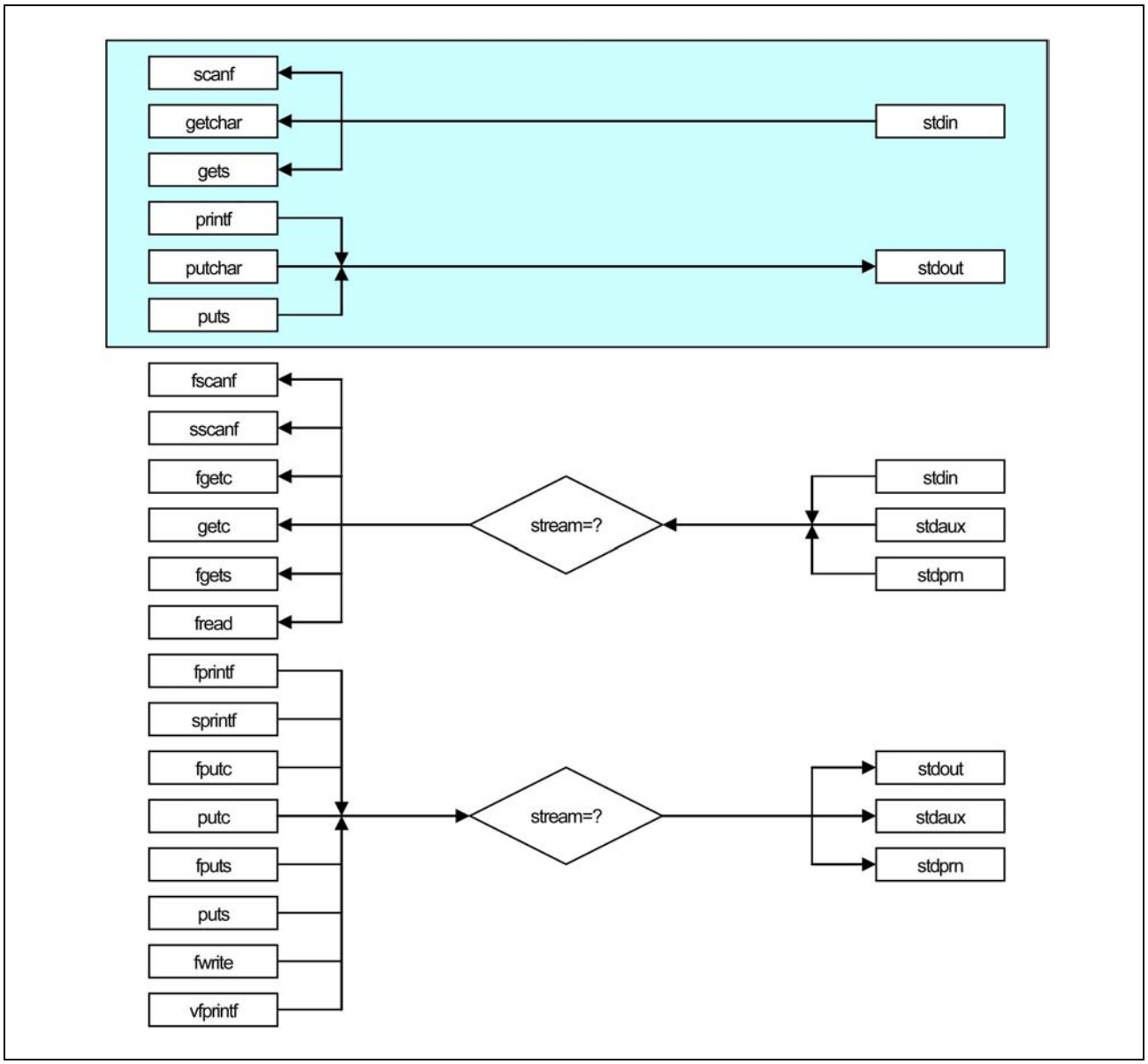
NC30 标准程序库具有五个作为外部结构的流数据项目 （stdin、 stdout、 stderr、 stdaux 和 stdprn）。这些外部结构在标准标题文件 stdio.h 中定义，并控制每个流的模式信息 （有标志显示是输入或输出流）和状态信息 （有标志显示是错误或 EOF）。

附表 E.15　流的信息

流的信息	名称
stdin	标准输入
stdout	标准输出
stderr	标准错误输出 （错误被输出到 stdout）
stdaux	标准辅助 I/O
stdprn	标准打印机输出

附图 E.3 中的阴影所显示的与 NC30 标准程序库函数对应的流，被固定为标准输入 (stdin) 和标准输出 (stdout) 。这些函数的流将无法更改。 stderr 的输出方向在 #define 中定义为 stdout。

只有将流的指针指定为 fgetc 和 fputc 等参数的函数可更改流。



附图 E.3　　函数与流的关系

附图 E.4 显示 stdio.h 中的流定义。

```
/*****************************************************************************
*
* 标准 I/O 标题文件
                :
            （已省略）
                :
typedef struct _iobuf {
            char        _buff;                          /* 存储 ungetc 的缓冲器 */              ← [1]
            int         _cnt;                           /* _buff(1 或 0) 中的字符串数量 */      ← [2]
            int         _flag;                          /* 标志 */                              ← [3]
            int         _mod;                           /* 模式 */                              ← [4]
            int         (*_func_in)(void);      /* 一字节输入函数的指针 */                      ← [5]
            int         (*_func_out)(int);      /* 一字节输出函数的指针 */                      ← [6]
} FILE;
#define     _IOBUF_DEF
                :
            （已省略）
                :
extern FILE _iob[];
#define     stdin       (&_iob[0])      /* 基本输入 */
#define     stdout      (&_iob[1])      /* 基本输出 */
#define     stdaux      (&_iob[2])      /* 基本辅助输入输出 */
#define     stdprn      (&_iob[3])      /* 基本打印见输出 */

#define     stderr      stdout                  /* NC 不支持 */

/*************************************************************
*
*************************************************************/
#define     _IOREAD  1              /* 只读标志 */
#define     _IOWRT   2              /* 只写标志 */
#define     _IOEOF   4              /* 文件末尾标志 */
#define     _IOERR   8              /* 错误标志 */
#define     _IORW    16             /* 读写标志 */
#define     _NFILE   4              /* 流编号 */
#define     _TEXT    1              /* 文本模式标志 */
#define     _BIN     2              /* 二进制模式标志 */

            （其余省略）
                :
```

附图 E.4　　stdio.h 中的流定义

参考附图 E.4 中所示的文件结构的元素。项目 [1] 到 [6] 与附图 E.4 中的 [1] 到 [6] 对应。

(1) char _buff

函数 scanf 和 fscanf 在输入时会提前读取一个字符。若该字符无用，则将调用 ungetc，然后将该字符存储在这个变量中。

若这个变量中已存在数据，输入函数将把该数据用作输入数据。

(2) int _cnt

存储 _buff 数据的计数 （0 或 1）。

(3) int _flag

存储只读标志 (_IOREAD)、只写标志 (_IOWRT)、读写标志 (_IORW)、文件末尾标志 (_IOEOF) 和错误标志 (_IOERR) 。

- _IOREAD、 _IOWRT、 _IORW
  这些标志将指定流的操作模式。它们会在流的初始化过程中进行设置。
- _IOEOF、 _IOERR
  这些标志将根据 I/O 函数是否到达 EOF 或发生错误来进行设置。

(4) int _mod

存储表示文本模式 (_TEXT) 和二进制模式 (_BIN) 的标志。

- 文本模式
  I/O 数据的回送和字符的转换。参考 fgetc 和 fputc 函数的源程序 （fget.c 和 fputc.c），以了解回送和字符转换的详情。
- 二进制模式
  不转换 I/O 数据。这些标志在流的初始化块中进行设置。

(5) int (*_func_in)()

当流处于只读模式 (_IOREAD) 或读写模式 (_IORW) 时，将存储第 3 级的输入函数指针。在其它模式中，则将存储 NULL 指针。

这项信息用于第 2 级输入函数对第 3 级输入函数的间接调用。

(6) int (*_func_out)()

当流处于只写模式 (_IOWRT) 时，将存储第 3 级的输出函数指针。若流可输入 （_IOREAD 或 _IORW）并处于文本模式，它将为回送存储第 3 级的输出函数指针。在其它模式中，则将存储 NULL 指针。

这项信息用于第 2 级输出函数对第 3 级输出函数的间接调用。

在流初始化块中为 char_buff 以外的所有元素设置值。NC30 产品封装随附的标准程序库文件会在函数 init 中初始化流，该函数调用于 ncrt0.a30 启动程序。

附图 E.5 中显示 init 函数的源程序。

```
#include <stdio.h>

FILE  _iob[4];

void   init( void );

void   init( void )
{
        stdin->_cnt = stdout->_cnt = stdaux->_cnt = stdprn->_cnt = 0;
        stdin->_flag = _IOREAD;
        stdout->_flag = _IOWRT;
        stdaux->_flag = _IORW;
        stdprn->_flag = _IOWRT;

        stdin->_mod = _TEXT;
        stdout->_mod = _TEXT;
        stdaux->_mod = _BIN;
        stdprn->_mod = _TEXT;

        stdin->_func_in = _sget;
        stdout->_func_in = NULL;
        stdaux->_func_in = _sget;
        stdprn->_func_in = NULL;

        stdin->_func_out = _sput;
        stdout->_func_out = _sput;
        stdaux->_func_out = _sput;
        stdprn->_func_out = _pput;

#ifdef UART0
        speed(_96, _B8, _PN, _S2);
#else /* UART1：默认值 */
        speed(_96, _B8, _PN, _S2);
#endif
        init_prn();
}
```

附图 E.5　　init 函数的源文件 (init.c)

在使用两个 M16C/60 系列 UART 的系统中，修改 init 函数如下。上一小节，我们在 device.c 源文件中将 UART0 函数暂时设置为 _sget0、 _sput0 和 speed0。

1. 为 UART0 流使用标准辅助 I/O （stdaux）。
2. 为标准辅助 I/O 设置标志 （_flag）和模式 （_mod），以符合系统的要求。
3. 为标准辅助 I/O 设置第 3 级函数指针。
4. 为 UART0 删除 speed 函数的条件编译命令，并更改为函数 speed0。

这些设置使两个 UART 都能被使用。不过，使用标准 I/O 流的函数，无法用于 UART0 所使用的标准辅助 I/O。因此，只能使用将流当作参数的函数。附图 E.6 中显示了更改 init 函数的方法。

```
void init( void )
{
                :
        （已省略）
                :
    stdaux->_flag = _IORW;              ← [2] （设置读写模式）
                :
        （已省略）
                :
    stdaux->_mod = _TEXT;               ← [2] （设置文本模式）
                :
        （已省略）
                :
    stdaux->_func_in = _sget0;          ← [3] （设置 UART0 第 3 级输入函数）
                :
        （已省略）
                :
    stdaux->_func_out = _sput0;         ← [3] （设置 UART0 第 3 级输入函数）
                :
        （已省略）
                :
    speed(_96, _B8, _PN, _S2);          ← [4] （设置 UART0 speed 函数）
    init_prn();
}

* [2] 到 [4] 与上述设置描述中的项目对应。
```

附图 E.6　　修改 init 函数

(c) 结合修改后的源程序

要在目标系统中结合修改后的源程序的方法有两种：

1. 在连接时指定修改后的函数源文件的目标文件。
2. 使用 NC30 产品封装随附的命令描述文件 （MS-Windows 下的 makefile.dos）来更新程序库文件。

在方法 [1] 中，连接时所指定的函数将生效，而程序库文件中具有相同名称的函数将被排除。

附图 E.7 中显示了方法 (1)。附图 E.8 则显示了方法 (2)。

```
% nc30  -c  -g  -osample  ncrt0.a30  device.r30  init.r30  sample.c<RET>

* 此范例显示在 device.c 和 init.c 修改后的合并的行。
```

附图 E.7 直接连接修改后的源程序的方法

```
% make  <RET>
```

附图 E.8 使用修改后的源程序来更新程序库的方法

附录 F　错误信息

本附录将描述本编译器所输出的错误信息和警告信息，以及它们的解决方法。

附录 F.1　信息格式

若本编译器在处理过程中发现到错误，它将在屏幕上显示一则错误信息，并停止编译。

下面显示错误信息和警告信息的格式。

```
nc30: [ 错误信息 ]
```

附图 F.1　编译驱动器错误信息的格式

```
[Error(cpp30. 错误编号 ): 文件名，行号 ] 错误信息
[Error(ccom): 文件名，行号 ] 错误信息
[Fatal(ccom): 文件名，行号 ] 错误信息　　← *1
```

附图 F.2　命令错误信息的格式

```
[Warning(cpp30. 警告编号 ): 文件名，行号 ] 警告信息
[Warning(ccom): 文件名，行号 ] 警告信息
```

附图 F.3　命令警告信息的格式

*1. 致命错误信息
此错误信息一般不输出。请联系最靠近的瑞萨办事处，并提供所显示的信息详情。

附录 F.2 nc30 错误信息

附表 F.1 和附表 F.2 列出了 nc30 编译驱动器的错误信息和它们的解决方法。

附表 F.1 nc30 错误信息 (1)

错误信息	描述和解决方法
Arg list too long （参数列表太长）	• 用于启动相应处理系统的命令行长于系统定义的字符串。 ⇒ 指定 NC30 选项以确保不超过系统定义的字符数。使用 -v 选项检查用于每个处理块的命令行。
Cannot analyze error （无法分析错误）	• 此错误信息一般不显示。（这是一项内部错误。） ⇒ 请联系瑞萨科技。
command-file line characters exceed 2048 （命令文件行字符超过 2048）	• 命令行的一行或多行有超过 2048 个字符。 ⇒ 将命令行中每行的字符数减少到最多 2048 个。
Core dump(command_name) （内存的信息转储 （命令名称））	• 处理系统 （在括号中指明）引起了内存的信息转储。 ⇒ 处理系统未正确运行。检查环境变量和包含处理系统的目录。若处理系统仍无法正确运行，请联系瑞萨科技。
Exec format error （执行格式错误）	• 损坏的处理系统可执行文件。 ⇒ 重新安装处理系统。
Ignore option '-?' （忽略选项 ‘-?’）	• 指定了不正确的选项 （-?）。 ⇒ 请指定正确的选项。
illegal option （不正确的选项）	• 为 -as30 或 -ln30 指定了超过 100 个字符的选项。 ⇒ 将选项减少到 99 个或更少的字符。
Invalid argument （无效的参数）	• 这是一项内部错误。（此错误信息一般不显示。） ⇒ 请联系瑞萨科技。
Invalid option '-?' （无效的选项 ‘-?’）	• 选项 “-?” 中未指定所需的参数。 ⇒ 在 “-?” 后指定所需的参数。 • 在 -? 选项及其参数之间指定了空格。 ⇒ 删除在 -? 选项及其参数之间的空格。
Invalid option '-o' （无效的选项 ‘-o’）	• -o 选项后未指定输出文件名。 ⇒ 指定输出文件的名称。不要指定文件扩展名。
Invalid suffix '.xxx' （无效的后缀 ‘.xxx’）	• 指定了 NC30 无法识别的文件名扩展名 （除 .c、 .i、 .a30、 .r30、 .x30 外）。 ⇒ 指定具有正确扩展名的文件名。

附表 F.2　　nc30 错误信息 (2)

错误信息	描述和解决方法
No such file or directory （无此类文件或目录）	• 处理系统将不会运行。 ⇒ 检查是否在环境变量中正确设置了处理系统的目录。
Not enough core （无足够内存）	• 内存的对换区不足。 ⇒ 增加内存的对换区。
Permission denied （权限被拒绝）	• 处理系统将不会运行。 ⇒ 检查处理系统的存取权限。或者，当存取权限无误时，检查是否在环境变量中正确设置了处理系统的目录。
Can't open command file （无法打开命令文件）	• 无法打开 ‘@’所指定的命令文件。 ⇒ 指定正确的输入文件。
too many options （太多选项）	• 此错误信息一般不显示。（这是一项内部错误。） ⇒ 编译选项的指定不可超过 99 个字符。
Result too large （结果太大）	• 这是一项内部错误。（此错误信息一般不显示。） ⇒ 请联系瑞萨科技。
Too many open files （打开的文件太多）	• 这是一项内部错误。（此错误信息一般不显示。） ⇒ 请联系瑞萨科技。

附录 F.3 cpp30 错误信息

附表 F.3 到附表 F.5 列出了 cpp30 预处理器所输出的错误信息和它们的解决方法。

附表 F.3 cpp30 错误信息 (1)

编号	错误信息	描述和解决方法
1	illegal command option （不正确的命令选项）	• 输入文件名被指定了两次。 ⇒ 只指定输入文件名一次。
		• 为输入和输出文件指定了相同的名称。 ⇒ 为输入和输出文件指定不同的名称。
		• 输出文件名被指定了两次。 ⇒ 只指定输出文件名一次。
		• 命令行以 -o 选项结尾。 ⇒ 在 -o 选项后指定输出文件的名称。
		• 指定包含文件路径的 -I 选项超过限制。 ⇒ 指定 -I 选项 8 次或更少次。
		• 命令行以 -I 选项结尾。 ⇒ 在 -I 选项后指定包含文件名称。
		• -D 选项后的字符串不是可以在宏名称中使用的字符类型 （字母或下划线）。不正确的宏名称定义。 ⇒ 正确指定宏名称，并正确定义宏。
		• 命令行以 -D 选项结尾。 ⇒ 在 -D 选项后指定宏文件名。
		• -U 选项后的字符串不是可以在宏名称中使用的字符类型 （字母或下划线）。 ⇒ 正确定义宏。
		• 在 cpp30 命令行上指定了不正确的选项。 ⇒ 只指定适当的选项。
11	cannot open input file （无法打开输入文件）	• 未找到输入文件。 ⇒ 指定正确的输入文件名。
12	cannot close input file （无法关闭输入文件）	• 无法关闭输入文件。 ⇒ 检查输入文件名。
14	cannot open output file （无法打开输出文件）	• 无法打开输出文件。 ⇒ 指定正确的输出文件名。
15	cannot close output file （无法关闭输出文件）	• 无法关闭输出文件。 ⇒ 检查磁盘上的可用空间。
16	cannot write output file （无法编写输出文件）	• 编写到输出文件时出现错误。 ⇒ 检查磁盘上的可用空间。

附表 F.4　　cpp30 错误信息 (2)

编号	错误信息	描述和解决方法
17	input file name buffer overflow （输入文件名缓冲器溢出）	• 输入文件名缓冲器已溢出。文件名中包括路径。 ⇒ 减少文件名和路径的长度 （使用 -I 选项指定标准目录）。
18	not enough memory for macro include file not found （存储器不足，找不到宏包含文件）	• 用于宏名称和宏内容的存储器不足。 ⇒ 增加存储器的对换区。
21	include file not found （未找到包含文件）	• 无法打开包含文件。 ⇒ 包含文件在当前目录及 -I 选项和环境变量指定的目录中。检查这些目录。
22	illegal file name error （不正确的文件名错误）	• 不正确的文件名。 ⇒ 指定正确的文件名。
23	include file nesting over （包含文件嵌套太多）	• 包含文件的嵌套超过限制 （40）。 ⇒ 将包含文件的嵌套减少到最多 8 级。
25	illegal identifier （不正确的标识符）	• #define 中出现错误。 ⇒ 正确为源文件编码。
26	illegal operation （不正确的操作）	• 预处理命令 #if - #elseif - #assert 运算表达式中的错误。 ⇒ 重新正确编写运算表达式。
27	macro argument error （宏参数错误）	• 扩展宏时宏参数数量出现错误。 ⇒ 检查宏定义和引用，并根据需要更正。
28	input buffer over flow （输入缓冲器溢出）	• 读取源文件时发生了输入行缓冲器溢出。或者，在转换宏时缓冲器溢出。 ⇒ 将源文件中的每行减少为最多 1023 个字符。如果预计会进行宏转换，则修改代码以便在转换后行不会超过 1023 个字符。
29	EOF in comment （注释中的 EOF）	• 注释中到达文件末尾。 ⇒ 更正源文件。
31	EOF in preprocess command （预处理命令中的 EOF）	• 预处理命令中到达文件末尾。 ⇒ 更正源文件。
32	unknown preprocess command （未知的预处理命令）	• 指定了未知的预处理命令。 ⇒ 在 CPP30 中只可以使用以下预处理命令： #include、 #define、 #undef、 #if、 #ifdef、 #ifndef、 #else、 #endif、 #elseif、 #line、 #assert、 #pragma、 #error
33	new_line in string （字符串中的新行）	• 字符常数或字符串常数中包括了新行代码。 ⇒ 更正程序。
34	string literal out of range 509 characters （字符串文字超过 509 个字符的范围）	• 字符串超过了 509 个字符。 ⇒ 将字符串减少到最多 509 个字符。
35	macro replace nesting over （宏替换嵌套太多）	• 宏嵌套超过了限制 （20）。 ⇒ 将嵌套级数降低到最多 20 个。
41	include file error （包含文件错误）	• #include 指令中出现错误。 ⇒ 请加以更正。

附表 F.5　　cpp30 错误信息 (3)

编号	错误信息	描述和解决方法
43	illegal id name （不正确的 id 名称）	• #define 命令中的以下宏名称或参数出现错误： __FILE__、 __LINE__、 __DATE__、 __TIME__ ⇒ 更正源文件。
44	token buffer over flow （标志缓冲器溢出）	• #define 的标志字符缓冲器溢出。 ⇒ 减少标志字符的数量。
45	illegal undef command usage （不正确的 undef 命令用法）	• #undef 中出现错误。 ⇒ 更正源文件。
46	undef id not found （未找到 undef id）	• 要在 #undef 中取消定义的以下宏名称未进行定义： __FILE__、 __LINE__、 __DATE__、 __TIME__ ⇒ 检查宏名称。
52	illegal ifdef / ifndef command usage （不正确的 ifdef / ifndef 命令用法）	• #ifdef 中出现错误。 ⇒ 更正源文件。
53	elseif / else sequence error （elseif / else 顺序错误）	• 使用了 #elseif 或 #else，但未使用 #if ～ #ifdef ～ #ifndef。 ⇒ 仅在 #if ～ #ifdef ～ #ifndef 之后使用 #elseif 或 #else。
54	endif not exist （endif 不存在）	• 没有匹配 #if ～ #ifdef ～ #ifndef 的 #endif。 ⇒ 将 #endif 添加到源文件。
55	endif sequence error （endif 顺序错误）	• 使用了 #endif，但未使用 #if ～ #ifdef ～ #ifndef。 ⇒ 仅在 #if ～ #ifdef ～ #ifndef 之后使用 #endif。
61	illegal line command usage （不正确的 line 命令使用）	• #line 中出现错误。 ⇒ 更正源文件。

附录 F.4 cpp30 警告信息

附表 F.6 中显示 cpp30 所输出的警告信息和它们的解决方法。

附表 F.6 cpp30 警告信息

编号	警告信息	描述和解决方法
81	reserved id used （使用了保留的 id）	• 尝试定义或取消定义由 cpp30 保留的以下宏名称之一： __FILE__、 __LINE__、 __DATE__、 __TIME__ ⇒ 使用不同的宏名称。
82	assertion warning （断言警告）	• #assert 运算表达式中的结果为 0。 ⇒ 检查运算表达式。
83	garbage argument （无用参数）	• 预处理命令后存在注释以外的其它字符。 ⇒ 在预处理命令后将字符指定为注释 （/* 字符串 */）。
84	escape sequence out of range for character （转义序列超出字符的范围）	• 字符常数或字符串常数中的转义序列超过了 255 个字符。 ⇒ 将转义序列减少到 255 个字符之内。
85	redefined （重新定义）	• 为之前定义的宏重新定义了不同的内容。 ⇒ 将该内容与之前定义中的内容进行对比。
87	/* within comment （注释内出现 /*）	• 注释中含有 /*。 ⇒ 不要嵌套注释。
88	Environment variable 'NCKIN' must be 'SJIS' or 'EUC' （环境变量 ‘NCKIN’ 必须是 ‘SJIS’ 或 ‘EUC’）	• 环境变量 ‘NCKIN’ 无效。 ⇒ 将 “SJIS” 或 “EUC” 设置到 NCKIN。
90	'Macro name' in #if is not defined, so it's treated as 0 （#if 中的 ‘宏名称’ 未定义，因此它被视为 0）	• 在 #if 中使用了未定义的宏名称。 ⇒ 检查宏的定义。

附录 F.5 ccom30 错误信息

附表 F.7 到附表 F.18 列出了 ccom30 编译器的错误信息和它们的解决方法。

附表 F.7 ccom30 错误信息 (1)

错误信息	描述和解决方法
#pragma PRAGMA-name functionname redefined （重新定义了 #pragma 杂注名 函数名称）	• 相同的函数在 #pragma 杂注名中定义了两次。 ⇒ 确保 #pragma 杂注名只被声明一次。
#pragma PRAGMA-name functionargument is long-long or double （#pragma 杂注名的函数参数为 long-long 或 double）	• 用于通过 “#pragma 程序名称 函数名称”指定的函数的参数为 long long 类型或 double 类型。 ⇒ long long 类型和 double 类型无法在通过 “#pragma 程序名称 函数名称”指定的函数中使用。请使用其它类型。
#pragma PRAGMA-name & function prototype mismatched （#pragma 杂注名和函数原型不匹配）	• #pragma 杂注名指定的函数不匹配原型声明中的参数内容。 ⇒ 确保它与原型声明中的参数匹配。
#pragma PRAGMA-name's function argument is struct or union （#pragma 杂注名的函数参数为 struct 或 union）	• 在 #pragma 杂注名所指定函数的原型声明中指定了 struct 或 union 类型。 ⇒ 在原型声明中指定 int 或 short 类型、 2 字节指针类型，或枚举类型。
#pragma PRAGMA-name must be declared before use （#pragma 杂注名必须在使用前声明）	• #pragma 杂注名声明中指定的函数在该函数调用后才定义。 ⇒ 在调用函数前声明函数。
#pragma BITADDRESS variable is not _Bool type （#pragma BITADDRESS 的变量不是 _Bool 类型）	• #pragma BITADDRESS 指定的变量不是 _Bool 类型。 ⇒ 使用 _Bool 类型来声明变量。
#pragma INTCALL function's argument on stack （#pragma INTCALL 函数的参数在堆栈上）	• #pragma INTCALL 中声明的函数主体用 C 语言编写时，参数通过堆栈传递。 ⇒ #pragma INTCALL 中声明的函数主体用 C 语言编写时，指定参数通过堆栈传递。
#pragma PARAMETER function's register not allocated （未分配 #pragma PARAMETER 函数的寄存器）	• 无法分配在 #pragma PARAMETER 所声明函数中指定的寄存器。 ⇒ 使用正确的寄存器。
'const' is duplicate （‘const’ 重复）	• const 被描述超过两次。 ⇒ 正确编写类型修饰符。
'far' & 'near' conflict （‘far’ 和 ‘near’ 冲突）	• far/near 被描述超过两次。 ⇒ 正确编写 near/far。
'far' is duplicate （‘far’ 重复）	• far 被描述超过两次。 ⇒ 正确编写 far。
'near' is duplicate （‘near’ 重复）	• near 被描述超过两次。 ⇒ 正确编写 near。
'static' is illegal storage class for agument （‘static’ 是不正确的参数存储类）	• 参数声明中使用了不正确的存储类。 ⇒ 使用正确的存储类。
'volatile' is duplicate （‘volatile’ 重复）	• volatile 被描述超过两次。 ⇒ 正确编写类型修饰符。

附表 F.8　　ccom30 错误信息 (2)

错误信息	描述和解决方法
(can't read C source from filename line number for error message) （错误信息造成无法从文件名行号读取 C 源）	• 源行有错误，因而无法显示。 找不到文件名所表示的文件，或者文件中不存在行号。 ⇒ 检查文件是否真的存在。
(can't open C source filename for error message) （错误信息造成无法打开 C 源文件名）	• 无法打开有错误的源文件。 ⇒ 检查文件是否存在。
argument type given both places （在两个位置都指定了参数类型）	• 函数定义中的参数声明与单独提供的参数列表重叠。 ⇒ 为这项参数声明选择参数列表或参数声明。
array of functions declared （所声明函数的数组）	• 数组声明中的数组类型定义为函数。 ⇒ 为数组类型指定标量类型 struct/union。
array size is not constant integer （数组大小不是整数常数）	• 数组声明中元素的数量不是常数。 ⇒ 使用一个常数描述元素的数量。
asm()'s string must have only 1 $b （asm() 的字符串必须仅有 1 个 $b）	• $b 在 asm 语句中被描述超过两次。 ⇒ 确保 $b 只描述一次。
asm()'s string must not have more than 3 $$ or $@ （asm() 的字符串不可包含超过 3 个 $$ 或 $@）	• $$ 或 $@ 在 asm 语句中被描述超过三次。 ⇒ 确保 $$ （$@）只描述两次。
auto variable's size is zero （auto 变量的大小为零）	• auto 存储区中声明了具有 0 个元素或无元素的数组。 ⇒ 更正代码。
bitfield width exceeded （位字段宽度超出）	• 位字段宽度超出数据类型的位宽度。 ⇒ 确保在位字段中声明的数据类型位宽度未超出。
bitfield width is not constant integer （位字段宽度不是整数常数）	• 位字段的位宽度不是一个常数。 ⇒ 使用一个常数来编写位宽度。
can't get bitfield address by '&' operator （无法通过 ‘&’ 运算符获得位字段地址）	• 位字段类型使用 & 运算符来编写。 ⇒ 不要使用 & 运算符编写位字段类型。
can't get inline function's address by '&' operator （无法通过 ‘&’ 运算符获得直接插入函数的地址）	• 在直接插入函数中编写了 & 运算符。 ⇒ 不要在直接插入函数中使用 & 运算符。
can't get size of bitfield （无法获得位字段大小）	• 位字段类型使用 sizeof 运算符来编写。 ⇒ 不要使用 sizeof 运算符编写位字段类型。
can't get void value （无法获得 void 值）	• 在赋值表达式右侧为 void 类型的情况下，尝试获得 void 类型数据。 ⇒ 检查数据类型。
can't output to file-name （无法输出至文件名）	• 无法写入该文件。 ⇒ 检查剩余的磁盘容量或文件的存取权限。
can't open file-name （无法打开文件名）	• 无法打开文件。 ⇒ 检查文件权限。
can't set argument （无法设置参数）	• 实际参数的类型与原型声明不匹配。无法在寄存器 （参数）中设置参数。 ⇒ 更正类型的不匹配。
case value is duplicated （case 值重复）	• case 的值被使用超过一次。 ⇒ 确保使用过一次的 case 值在一个 switch 语句内不会再被使用。
conflict declare of variable-name （变量名称的声明冲突）	• 变量通过不同的存储类定义了两次。 ⇒ 使用相同的存储类来为变量声明两次。

附表 F.9　　ccom30 错误信息 (3)

错误信息	描述和解决方法
conflict function argument type of variable-name （变量名称的函数参数类型冲突）	• 参数列表包含相同的变量名称。 ⇒ 更改变量名称。
declared register parameter function's body declared （所声明的寄存器参数函数的主体已声明）	• 通过 #pragma PARAMETER 声明的函数的主体用 C 语言定义。 ⇒ 不要用 C 语言为函数定义主体。
default function argument conflict （默认函数参数冲突）	• 参数的默认值在原型声明中被声明超过两次。 ⇒ 确保参数的默认值只声明一次。
default: is duplicated （默认值重复）	• 默认值被使用超过一次。 ⇒ 在一个 switch 语句内只使用一个默认值。
do while (struct/union) statement （do while （struct/union）语句）	• do-while 语句的表达式中使用了 struct 或 union 类型。 ⇒ 在 dowhile 语句中为表达式使用标量类型。
do while (void)statement （do while （void）语句）	• do while 语句的表达式中使用了 void 类型。 ⇒ 在 do while 语句中为表达式使用标量类型。
duplicate frame position defind variable-name （在重复的帧位置定义了变量名称）	• auto 变量被描述超过两次。 ⇒ 正确编写类型说明符。
Empty declare （空的声明）	• 只找到一个存储类和类型说明符。 ⇒ 编写声明符。
float and double not have sign （float 和 double 无符号）	• 在 float 或 double 中描述了带符号 / 无符号的说明符。 ⇒ 正确编写类型说明符。
floating point value overflow （浮点值溢出）	• 浮点立即值超出可表达的范围。 ⇒ 确保值在范围内。
floating type's bitfield （浮点类型的位字段）	• 声明了无效类型的位字段。 ⇒ 使用整数类型来声明位字段。
for (; struct/union ;) statement （for （; struct/union ;）语句）	• for 语句的第二表达式中使用了 struct 或 union 类型。 ⇒ 使用标量类型描述 for 语句的第二表达式。
for (; void ;) statement （for （; void ;）语句）	• for 语句的第二表达式具有 void。 ⇒ 使用标量类型作为 for 语句的第二表达式。
function initialized （函数已初始化）	• 为函数声明描述了初始化表达式。 ⇒ 删除初始化表达式。
function member declared （函数成员已声明）	• 一个 struct 或 union 的成员是函数类型。 ⇒ 正确编写成员。
function returning a function declared （函数返回已声明的函数）	• 函数声明中返回值的类型是函数类型。 ⇒ 将类型更改为 “pointer to function”（指向函数的指针）等。
function returning an array （函数返回数组）	• 函数声明中返回值的类型是数组类型。 ⇒ 将类型更改为 “pointer to function”（指向函数的指针）等。
handler function called （已调用处理程序函数）	• 调用了 #pragma HANDLER 指定的函数。 ⇒ 谨慎处理以避免调用处理程序。
identifier (variable-name) is duplicated （标识符（变量名称）重复）	• 变量被定义超过一次。 ⇒ 正确指定变量定义。

附表 F.10　　ccom30 错误信息 (4)

错误信息	描述和解决方法
if (struct/union) statement （if （struct/union）语句）	• if 语句的表达式中使用了 struct 或 union 类型。 ⇒ 表达式必须具有标量类型。
if (void) statement （if （void）语句）	• if 语句的表达式中使用了 void 类型。 ⇒ 表达式必须具有标量类型。
illegal storage class for argument, 'inline' ignored （不正确的参数存储类，‘inline’将被忽略）	• 在函数内的声明语句中声明了直接插入函数。 ⇒ 在函数外进行声明。
illegal storage class for argument, 'interrupt' ignored （不正确的参数存储类，‘interrupt’将被忽略）	• 在函数内的声明语句中声明了中断函数。 ⇒ 在函数外进行声明。
incomplete array access （不完整的数组存取）	• 尝试引用不完整的数组。 ⇒ 定义数组的大小。
incomplete return type （不完整的返回类型）	• 尝试引用类型不完整的返回变量。 ⇒ 检查返回变量。
incomplete struct get by [] （[] 所获取的不完整 struct）	• 尝试对没有已定义成员的不完整 struct 或 union 数组进行了引用或初始化。 ⇒ 先定义完整的 struct 或 union。
incomplete struct member （不完整的 struct 成员）	• 尝试引用不完整的 struct 成员。 ⇒ 先定义完整的 struct 或 union。
incomplete struct initialized （对不完整的 struct 进行了初始化）	• 尝试对没有已定义成员的不完整 struct 或 union 数组进行了初始化。 ⇒ 先定义完整的 struct 或 union。
incomplete struct return function call （不完整的 struct 返回函数调用）	• 尝试调用了不完整 struct 或 union 的返回函数，且 struct 或 union 没有已定义的成员。 ⇒ 先定义一个完整的 struct 或 union。
incomplete struct / union's member access （不完整的 struct/union 成员存取）	• 尝试引用了没有已定义成员的不完整的 struct 或 union 成员。 ⇒ 先定义一个完整的 struct 或 union。
incomplete struct / union(tagname)'s member access （不完整的 struct/union （标签名称）成员存取）	• 尝试引用了没有已定义成员的不完整的 struct 或 union 成员。 ⇒ 先定义一个完整的 struct 或 union。
inline function have invalid argument or return code （具有无效参数或返回代码的直接插入函数）	• 直接插入函数具有无效的参数或无效的返回值。 ⇒ 正确编写参数或无效的返回值。
inline function is called as normal function before （直接插入函数曾作为正常函数调用）	• 直接插入存储类中声明的函数曾作为普通函数调用。 ⇒ 总是确保在使用直接插入函数前先进行定义。
inline function's address used （所使用的直接插入函数的地址）	• 尝试引用了直接插入函数的地址。 ⇒ 不要使用直接插入函数的地址。
inline function's body is not declared previously （直接插入函数的主体未声明）	• 未定义直接插入函数的主体。 ⇒ 使用直接插入函数时，在函数调用前先定义函数主体。
inline function (function-name) is recursion （直接插入函数 （函数名称）是递归）	• 无法进行对直接插入函数的递归调用。 ⇒ 在使用直接插入函数时，请勿使用递归。

附表 F.11　　ccom30 错误信息 (5)

错误信息	描述和解决方法
interrupt function called （已调用中断函数）	• 调用了 #pragma INTERRUPT 指定的函数。 ⇒ 谨慎处理以避免调用中断处理函数。
invalid environment variable: (environment variable -name) （无效的环境变量：（环境变量名称）	• 环境变量 NCKIN/NCKOUT 中指定的环境变量，不是使用 SJIS 和 EUC 指定。 ⇒ 检查所使用的环境变量。
invalid function default argument （无效的函数默认参数）	• 函数的默认参数不正确。 ⇒ 在具有默认参数的函数原型声明和函数定义段不匹配时，便会发生这种错误。确保它们匹配。
invalid push （无效的 push）	• 尝试推进 void 类型作为函数参数等。 ⇒ 无法推进 void 类型。
invalid '? :' operand （无效的 ‘? : ’操作数）	• ?：运算存在错误。 ⇒ 检查每个表达式。也请留意：左侧和右侧的表达式必须属于相同类型。
invalid '!=' operands （无效的 ‘!=’操作数）	• != 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '&&' operands （无效的 ‘&&’操作数）	• && 运算存在错误。 • 检查运算符左侧和右侧的表达式。
invalid '&' operands （无效的 ‘&’操作数）	• & 运算存在错误。 ⇒ 检查运算符右侧的表达式。
invalid '&=' operands （无效的 ‘&=’操作数）	• &= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '( )' operand （无效的 ‘( )’操作数）	• ( ) 左侧的表达式不是函数。 ⇒ 在 ( ) 的左侧表达式中编写函数或指向函数的指针。
invalid '*' operands （无效的 ‘*’操作数）	• 如果是乘法，则是 * 运算存在错误。 如果 * 是指针运算符，则右侧的表达式不是指针类型。 ⇒ 对于乘法，检查运算符左侧和右侧的表达式。对于指针，检查右侧表达式的类型。
invalid '*=' operands （无效的 ‘*=’操作数）	• *= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '+' operands （无效的 ‘+’操作数）	• + 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '+=' operands （无效的 ‘+=’操作数）	• += 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '-' operands （无效的 ‘-’操作数）	• - 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '-=' operands （无效的 ‘-=’操作数）	• -= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。

附表 F.12　　ccom30 错误信息 (6)

错误信息	描述和解决方法
invalid '/=' operands （无效的 ‘/=’ 操作数）	• /= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '<<' operands （无效的 ‘<<’ 操作数）	• << 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '<<=' operands （无效的 ‘<<=’ 操作数）	• <<= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '<=' operands （无效的 ‘<=’ 操作数）	• <= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '=' operand （无效的 ‘=’ 操作数）	• = 运算有错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '==' operands （无效的 ‘==’ 操作数）	• == 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '>=' operands （无效的 ‘>=’ 操作数）	• >= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '>>' operands （无效的 ‘>>’ 操作数）	• >> 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '>>=' operands （无效的 ‘>>=’ 操作数）	• >>= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '[ ]' operands （无效的 ‘[ ]’ 操作数）	• [ ] 的左侧表达式不是数组类型或指针类型。 ⇒ 使用数组或指针类型编写 [ ] 的左侧表达式。
invalid '^=' operands （无效的 ‘^=’ 操作数）	• ^= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '|=' operands （无效的 ‘|=’ 操作数）	• |= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '||' operands （无效的 ‘||’ 操作数）	• || 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid '%=' operands （无效的 ‘%=’ 操作数）	• %= 运算存在错误。 ⇒ 检查运算符左侧和右侧的表达式。
invalid ++ operands （无效的 ++ 操作数）	• ++ 一元运算符或后缀运算符存在错误。 ⇒ 对于一元运算符，检查右侧的表达式。 对于后缀运算符，检查左侧的表达式。
invalid -- operands （无效的 -- 操作数）	• -- 一元运算或后缀运算存在错误。 ⇒ 对于一元运算符，检查右侧的表达式。 对于后缀运算符，检查左侧的表达式。
invalid -> used （使用了无效的 ->）	• -> 的左侧表达式不是 struct 或 union。 ⇒ -> 的左侧表达式必须具有 struct 或 union。

附表 F.13　　ccom30 错误信息 (7)

错误信息	描述和解决方法
invalid (?;)'s condition （无效的 （?;）条件）	• 三元运算符被错误地编写。 ⇒ 检查三元运算符。
Invalid #pragma OS Extended function interrupt number （无效的 #pragma OS 扩展函数中断编号）	• #pragma OS 扩展函数中的 INT 编号无效。 ⇒ 正确指定该编号。
Invalid #pragma INTCALL interrupt number （无效的 #pragma INTCALL 中断编号）	• #pragma INTCALL 中的 INT 编号无效。 ⇒ 正确指定该编号。
Invalid #pragma SPECIAL special page number （无效的 #pragma SPECIAL 特殊页编号）	• #pragma SPECIAL 中的编号无效。 ⇒ 正确指定该编号。
invalid CAST operand （无效的 CAST 操作数）	• cast 运算存在错误。void 类型无法转换类型到任何其它类型；它既不可以从 struct 或 union 类型转换，也不可以转换类型到 struct 或 union 类型。 ⇒ 正确编写表达式。
invalid asm()'s argument （无效的 asm() 参数）	• asm 语句中可使用的变量只有 auto 变量和参数。 ⇒ 为该语句使用 auto 变量或参数。
invalid bitfield declare （无效的位字段声明）	• 位字段声明存在错误。 ⇒ 正确编写声明。
invalid break statements （无效的 break 语句）	• break 语句放置在无法使用之处。 ⇒ 确保它编写在 switch、 while、 do-while 和 for 中。
invalid case statements （无效的 case 语句）	• switch 语句存在错误。 ⇒ 正确编写 switch 语句。
invalid case value （无效的 case 值）	• case 值存在错误。 ⇒ 编写 int 类型或 enum 类型的常数。
invalid cast operator （无效的类型转换运算符）	• 类型转换运算符的使用不正确。 ⇒ 正确编写表达式。
invalid continue statements （无效的 continue 语句）	• continue 语句放置在无法使用之处。 ⇒ 在 while、 do-while 和 for 块中使用它。
invalid default statements （无效的 default 语句）	• switch 语句存在错误。 ⇒ 正确编写 switch 语句。
invalid enumerator initialized （初始化了无效的枚举）	• 例如，通过编写变量名称，错误地指定了枚举的初始值。 ⇒ 正确编写枚举的初始值。
invalid function argument （无效的函数参数）	• 在函数定义的参数定义中声明了未包括在参数列表中的参数。 ⇒ 声明包括在参数列表中的参数。
invalid function's argument declaration （无效的函数参数声明）	• 函数的参数被错误地声明。 ⇒ 进行正确的编写。
invalid function declare （无效的函数声明）	• 函数定义存在错误。 ⇒ 检查有错误的行或检查前一个函数定义。
invalid initializer （无效的初始化表达式）	• 初始化表达式存在错误。 此错误包括括号过多、太多的初始化表达式、函数中由 auto 变量所初始化的 static 变量，或另一变量所初始化的变量。 ⇒ 正确编写初始化表达式。
invalid initializer of variable-name （无效的变量名称的初始化表达式）	• 初始化表达式存在错误。 例如，此错误包括通过变量描述的位字段初始化表达式。 ⇒ 正确编写初始化表达式。

附表 F.14　　ccom30 错误信息 (8)

错误信息	描述和解决方法
invalid initializer on array （无效的数组初始化表达式）	• 初始化表达式存在错误。 ⇒ 检查括号中的初始化表达式数量是否与数组元素数量和结构成员数量匹配。
invalid initializer on char array （无效的 char 数组初始化表达式）	• 初始化表达式存在错误。 ⇒ 检查括号中的初始化表达式数量是否与数组元素数量和结构成员数量匹配。
invalid initializer on scalar （无效的标量初始化表达式）	• 初始化表达式存在错误。 ⇒ 检查括号中的初始化表达式数量是否与数组元素数量和结构成员数量匹配。
invalid initializer on struct （无效的 struct 初始化表达式）	• 初始化表达式存在错误。 ⇒ 检查括号中的初始化表达式数量是否与数组元素数量和结构成员数量匹配。
invalid initializer, too many brace （无效的初始化表达式，太多括号）	• 在 auto 存储类的标量类型初始化表达式中使用了太多的括号 { }。 ⇒ 减少使用括号 { } 的次数。
invalid lvalue （无效的 lvalue）	• 赋值语句的左侧不是 lvalue。 ⇒ 在语句的左侧编写可替代的表达式。
invalid lvalue at '=' operator （‘=’运算符上的 lvalue 无效）	• 赋值语句的左侧不是 lvalue。 ⇒ 在语句的左侧编写可替代的表达式。
invalid member （无效的成员）	• 成员引用存在错误。 ⇒ 进行正确的编写。
invalid member used （使用了无效的成员）	• 成员引用存在错误。 ⇒ 进行正确的编写。
invalid redefined type name of (identifier) （重新定义的 （标识符）类型名称无效）	• 相同的标识符在 typedef 中被定义超过一次。 ⇒ 正确编写标识符。
invalid return type （无效的返回类型）	• 函数的返回值类型不正确。 ⇒ 进行正确的编写。
invalid sign specifier （无效的符号说明符）	• 带符号 / 无符号的说明符被描述了两次或更多次。 ⇒ 正确编写类型说明符。
invalid storage class for data （无效的数据存储类）	• 存储类被错误地指定。 ⇒ 进行正确的编写。
invalid struct or union type （无效的 struct 或 union 类型）	• 为枚举类型的数据引用了结构或联合成员。 ⇒ 进行正确的编写。
invalid truth expression （无效的 truth 表达式）	• 在条件表达式 （?:）的第一个表达式中使用了 void、 struct 或 union 类型。 ⇒ 使用标量类型来编写此表达式。
invalid type specifier （无效的类型说明符）	• 相同类型的说明符在 “int int i;” 中被描述了两次或更多次，或者不兼容的类型标识符在 “float int i;” 中被描述。 ⇒ 正确编写类型说明符。
invalid type's bitfield （无效类型的位字段）	• 声明了无效类型的位字段。 ⇒ 对位字段使用整数类型。
invalid type specifier,long long long （无效的类型说明符， long long long）	• 说明符 “long” 被描述三次或更多次。 ⇒ 检查类型。

附表 F.15　　ccom30 错误信息 (9)

错误信息	描述和解决方法
invalid unary '!' operands （无效的一元 ‘!’ 操作数）	• ! 一元运算符的使用不正确。 ⇒ 检查运算符的右侧表达式。
invalid unary '+' operands （无效的一元 ‘+’ 操作数）	• + 一元运算符的使用不正确。 ⇒ 检查运算符的右侧表达式。
invalid unary '-' operands （无效的一元 ‘-’ 操作数）	• - 一元运算符的使用不正确。 ⇒ 检查运算符的右侧表达式。
invalid unary '~' operands （无效的一元 ‘～’ 操作数）	• ～ 一元运算符的使用不正确。 ⇒ 检查运算符的右侧表达式。
invalid void type （无效的 void 类型）	• void 类型说明符与 long 或 signed 一起使用。 ⇒ 正确编写类型说明符。
invalid void type, int assumed （无效的 void 类型，假设为 int）	• 不可声明 void 类型变量。它将被假设为 int 类型继续处理。 ⇒ 正确编写类型说明符。
invalid size of bitfield （无效的位字段大小）	• 获取位字段大小。 ⇒ 不在这个声明中编写位字段。
invalid switch statement （无效的 switch 语句）	• switch 语句的使用不正确。 ⇒ 进行正确的编写。
label label redefine （重新定义了 label label）	• 相同的标签在一个函数内被定义两次。 ⇒ 更改两个标签中其中一个的名称。
long long type's bitfield （long long 类型的位字段）	• 用 long long 类型指定位字段。 ⇒ 无法为位字段指定 long long 类型。
mismatch prototyped parameter type （已声明原型的参数类型不匹配）	• 参数类型不是在原型声明中声明的类型。 ⇒ 检查参数类型。
No #pragma ENDASM （无 #pragma ENDASM）	• #pragma ASM 不存在匹配的 #pragma ENDASM。 ⇒ 编写 #pragma ENDASM。
No declarator （无声明符）	• 声明语句不完整。 ⇒ 编写完整的声明语句。
Not enough memory （存储器不足）	• 存储区不足。 ⇒ 为 Windows 增加存储器或虚拟存储器。
not have 'long char' （不存在 ‘long char’ 类型）	• 同时使用了类型说明符 long 和 char。 ⇒ 正确编写类型说明符。
not have 'long float' （不存在 ‘long float’ 类型）	• 同时使用了类型说明符 long 和 float。 ⇒ 正确编写类型说明符。
not have 'long short' （不存在 ‘long short’）	• 同时使用了类型说明符 long 和 short。 ⇒ 正确编写类型说明符。
not static initializer for variablename （不是用于变量名称的 static 初始化表达式）	• static 变量的初始化表达式存在错误。例如，这是因为初始化表达式是一个函数调用。 ⇒ 正确编写初始化表达式。
not struct or union type （不是 struct 或 union 类型）	• -> 的左侧表达式不是 struct 或 union 类型。 ⇒ 使用 struct 或 union 类型描述 -> 的左侧表达式。
redeclare of variable-name （重新声明了变量名称）	• 一个变量名称被声明了两次。 ⇒ 更改两个变量名称中其中一个的名称。
redeclare of enumerator （重新声明了枚举）	• 一个枚举被声明了两次。 ⇒ 更改两个枚举中其中一个的名称。

附表 F.16　　ccom30 错误信息 (10)

错误信息	描述和解决方法
redefine function function-name （重新定义了函数函数名称）	• 函数名称表示的函数被定义了两次。 ⇒ 函数只可定义一次。更改两个函数中任何一个的名称。
redefinition tag of enum tag-name （重新定义了 enum 标签名称的标志）	• 枚举被定义了两次。 ⇒ 确保枚举只定义一次。
redefinition tag of struct tag-name （重新定义了 struct 标签名称的标志）	• 结构被定义了两次。 ⇒ 确保结构只定义一次。
redefinition tag of union tag-name （重新定义了 union 标签名称的标志）	• 联合被定义了两次。 ⇒ 确保联合只定义一次。
reinitialized of variable-name （重新初始化了变量名称）	• 为同一变量指定了两次初始化表达式。 ⇒ 只指定一次初始化表达式。
restrict is duplicate （重复了限制）	• 限制被定义了两次。 ⇒ 确保限制只定义一次。
size of incomplete array type （不完整数组类型的大小）	• 尝试查找了大小未知的数组的 sizeof。这是无效的大小。 ⇒ 指定数组的大小。
size of incomplete type （不完整类型的大小）	• 在 sizeof 运算符的操作数中使用了未定义的结构或联合。 ⇒ 先定义结构或联合。 • 定义为 sizeof 运算符的操作数数组所具有的元素数量未知。 ⇒ 先定义结构或联合。
size of void （void 的大小）	• 尝试查找了 void 的大小。这是无效的大小。 ⇒ 找不到 void 的大小。
Sorry, stack frame memory exhaust, max. 64(or 255) bytes but now nnn bytes （抱歉，堆栈帧存储器已用完，最多 64 （或 255）字节，但现在是 nnn 字节）	• 堆栈帧内最多只可保留 128 字节的参数。目前，已使用了 nnn 字节。 ⇒ 减少参数的大小或数量。
Sorry, compilation terminated because of these errors in functionname （抱歉，由于函数名称中的这些错误，编译已终止。）	• 函数名称所示的某一函数中发生了错误。编译已终止。 ⇒ 在输出此信息前先更正所检测到的错误。
Sorry, compilation terminated because of too many errors （抱歉，由于太多的错误，编译已终止。）	• 源文件中的错误超出了上限 （50 个错误）。 ⇒ 在输出此信息前先更正所检测到的错误。
struct or enum's tag used for union （为 union 使用了 struct 或 enum 的标志）	• 为联合类型的标志名称使用了结构和枚举类型的标志名称。 ⇒ 更改标志名称。
struct or enum's tag used for enum （为 enum 使用了 struct 或 union 的标志）	• 为枚举类型的标志名称使用了结构和联合类型的标志名称。 ⇒ 更改标志名称。
struct or union,enum does not have long or sign （struct、 union 或 enum 不具有 long 或 sign）	• 为 struct/union/enum 类型说明符使用了类型说明符 long 或 signed。 ⇒ 正确编写类型说明符。

附表 F.17　　ccom30 错误信息 (11)

错误信息	描述和解决方法
switch's condition is floating （switch 的条件是浮点）	• 为 switch 语句的表达式使用了浮点类型。 ⇒ 使用整数类型或枚举类型。
switch's condition is void （switch 的条件是 void）	• 为 switch 语句的表达式使用了 void 类型。 ⇒ 使用整数类型或枚举类型。
switch's condition must integer （switch 的条件必须为整数）	• 为 switch 语句的表达式使用了除整数和枚举类型以外的其它无效类型。 ⇒ 使用整数类型或枚举类型。
syntax error （语法错误）	• 这是一项语法错误。 ⇒ 正确编写描述。
System Error （系统错误）	• 这是一项内部错误。（一般情况下不会出现。）此错误可能由发生在之前的其中一项错误所引起。 ⇒ 若在清除发生在之前的所有错误之后，这个错误仍然存在，请联系并将错误信息的内容发送给瑞萨科技。
too big data-length （数据长度太大）	• 尝试获取超过 32 位范围的地址。 ⇒ 确保将值设置在单片机所使用的地址范围内。
too big address （地址范围太大）	• 尝试设置了超过 32 位范围的地址。 ⇒ 确保将值设置在单片机所使用的地址范围内。
too many storage class of typedef （typedef 的存储类太多）	• extern/typedef/static/auto/register 等存储类说明符在声明中被描述超过两次。 ⇒ 不要描述存储类说明符超过两次。
type redeclaration of variable-name （重新声明了变量名称的类型）	• 每次使用不同的类型对变量进行了定义。 ⇒ 声明变量两次时总是使用相同的类型。
typedef initialized （typedef 已被初始化）	• 在通过 typedef 声明的变量中描述了初始化表达式。 ⇒ 删除初始化表达式。
uncomplete array pointer operation （不完整的数组指针运算）	• 对指针存取了不完整的多维数组。 ⇒ 指定多维数组的大小。
undefined label "label" used （使用了未定义的标签 “label”）	• 函数中用于 goto 的跳转地址标签未定义。 ⇒ 定义函数中的跳转地址标签。
union or enum's tag used for struct （为 struct 使用了 union 或 enum 的标志）	• 为结构类型的标志名称使用了联合与枚举类型的标志名称。 ⇒ 更改标志名称。
unknown function argument variable-name （未知的函数参数变量名称）	• 指定了未包括在参数列表中的参数。 ⇒ 检查参数。
unknown member "member-name" used （使用了未知的成员 “成员名称”）	• 所引用的成员未注册为任何结构或联合成员。 ⇒ 检查成员名称。

附表 F.18　　ccom30 错误信息 (12)

错误信息	描述和解决方法
unknown pointer to structure identifier "variable-name" （未知的结构标识符 “变量名称” 指针）	• -> 的左侧表达式不是 struct 或 union 类型。 ⇒ 使用 struct 或 union 作为 -> 的左侧表达式。
unknown size of struct or union （未知的 struct 或 union 大小）	• 使用了不确定大小的结构或联合。 ⇒ 在声明结构或联合变量前先声明结构或联合。
unknown structure identifier "variable-name" （未知的结构标识符 “变量名称”）	• “.” 的左侧表达式没有 struct 或 union。 ⇒ 使用 struct 或 union。
unknown variable "variable-name" used in asm() （在 asm() 中使用的未知的变量 “变量名称”）	• asm 语句中使用了未定义的变量名称。 ⇒ 定义变量。
unknown variable variable-name （未知的变量 变量名称）	• 使用了未定义的变量名称。 ⇒ 定义变量。
unknown variable variable-name used （使用了未知的变量变量名称）	• 使用了未定义的变量名称。 ⇒ 定义变量。
void array is invalid type ,int array assumed （void 数组是无效的类型，假设为 int 数组）	• 数组不可被声明为 void。将假设它是 int 类型继续处理。 ⇒ 正确编写类型说明符。
void value can't return （不能返回 void 值）	• 通过 cast 转换为 void 的值被用作了函数的返回值。 ⇒ 进行正确的编写。
while (struct/union) statement （while （struct/union）语句）	• while 语句表达式中使用了 struct 或 union。 ⇒ 使用标量类型。
while (void)statement （while （void）语句）	• while 语句表达式中使用了 void。 ⇒ 使用标量类型。
multiple #pragma EXT4MPTR's pointer, ignored （多个 #pragma EXT4MPTR 的指针，将被忽略）（仅限于 NC30）	• # pragma EXT4MPTR 被声明超过两次。 ⇒ 请勿声明 #pragma EXT4MPTR 超过两次。
zero size array member （零大小的数组成员）	• 数组的大小是零。 ⇒ 声明数组的大小。 • 结构成员包括了大小为零的数组。 ⇒ 零大小的数组无法作为结构的成员。
‘function-name' is recursion, then inline is ignored （‘函数名称’ 是递归调用，所以直接插入将被忽略）	• 声明直接插入的 ‘函数名称’ 被递归调用。直接插入的声明将被忽略。 ⇒ 更正语句，以便不递归调用此函数名称。

附录 F.6 ccom30 警告信息

附表 F.19 到附表 F.27 列出了 ccom30 编译器的警告信息和它们的解决方法。

附表 F.19 ccom30 警告信息 (1)

警告信息	描述和解决方法
#pragma pragma-name & HANDLER both specified （同时指定了 #pragma 杂注名和 HANDLER）	• 在一个函数中同时指定了 #pragma 杂注名和 #pragma HANDLER。 ⇒ #pragma 杂注名和 #pragma HANDLER 不能同时指定。
#pragma pragma-name & INTERRUPT both specified （同时指定了 #pragma 杂注名和 INTERRUPT）	• 在一个函数中同时指定了 #pragma 杂注名和 #pragma INTERRUPT。 ⇒ #pragma 杂注名和 #pragma INTERRUPT 不能同时指定。
#pragma pragma-name & TASK both specified （同时指定了 #pragma 杂注名和 TASK）	• 在一个函数中同时指定了 #pragma 杂注名和 #pragma TASK。 ⇒ #pragma 杂注名和 #pragma TASK 不能同时指定。
#pragma pragma-name format error （#pragma 杂注名的格式错误）	• #pragma 杂注名的编写错误。处理将继续。 ⇒ 进行正确的编写。
#pragma pragma-name format error, ignored （#pragma 杂注名的格式错误，将被忽略）	• #pragma 杂注名的编写错误。这一行将被忽略。 ⇒ 进行正确的编写。
#pragma pragma-name not function, ignored （#pragma 杂注名不是函数，将被忽略）	• 在 #pragma 杂注名中编写的名称不是函数。 ⇒ 使用函数名称来编写。
#pragma pragma-name's function must be predeclared, ignored （#pragma 杂注名的函数必须预先声明，将被忽略）	• 未声明 #pragma 杂注名中指定的函数。 ⇒ 对 #pragma 杂注名中指定的函数事先编写原型声明。
#pragma pragma-name's function must be prototyped, ignored （#pragma 杂注名的函数必须声明原型，将被忽略）	• #pragma 杂注名中指定的函数未声明原型。 ⇒ 对 #pragma 杂注名中指定的函数事先编写原型声明。
#pragma pragma-name's function return type invalid,ignored （#pragma 杂注名的函数返回类型无效，将被忽略）	• #pragma 杂注名中指定的函数的返回值类型无效。 ⇒ 确保返回值的类型为 struct、union 或 double 以外的任何其它类型。
#pragma pragma-name unknown switch,ignored （#pragma 杂注名未知的切换，将被忽略）	• #pragma 杂注名中指定的切换无效。 ⇒ 进行正确的编写。
#pragma pragma-name variable initialized, initialization ignored （#pragma 杂注名变量已初始化，初始化将被忽略）	• #pragma 杂注名中指定的变量已初始化。#pragma 杂注名的指定将失效。 ⇒ 删除 #pragma 杂注名或初始化表达式。
#pragma ASM line too long, then cut （#pragma ASM 行太长，因此被截断）	• 编写 #pragma ASM 的行超过了允许的 1,024 个字节的字符数。 ⇒ 将它编写在 1,024 个字节内。

附表 F.20　　ccom30 警告信息 (2)

警告信息	描述和解决方法
#pragma directive conflict （#pragma 杂注指定冲突）	• 为一个函数指定了不同函数的 #pragma。 ⇒ 进行正确的编写。
#pragma DMAC duplicate （#pragma DMAC 重复）（仅限于 NC308）	• 相同的 #pragma DMAC 被定义了两次。 ⇒ 请勿定义 #pragma DMAC 两次或更多次。
#pragma DMAC variable must be far pointer for variable-name, ignored （变量名称的 #pragma DMAC 变量必须是 far 指针，将被忽略）（仅限于 NC308）	• #pragma DMAC 所声明的变量必须是 far 指针。 DMAC 的声明将被忽略。 ⇒ 进行正确的编写。
#pragma DMAC variable must be unsigned int for variable-name, ignored （变量名称的 #pragma DMAC 变量必须是 unsigned int，将被忽略）（仅限于 NC308）	• #pragma DMAC 所声明的变量必须是 unsigned int 类型。 DMAC 的声明将被忽略。 ⇒ 进行正确的编写。
#pragma DMAC's variable must be pre-declared, ignored （#pragma DMAC 的变量必须预先声明，将被忽略）（仅限于 NC308）	• #pragma DMAC 所声明的变量必须声明类型。 ⇒ 进行正确的编写。
#pragma DMAC, register conflict （#pragma DMAC，寄存器冲突）（仅限于 NC308）	• 将多个变量分配到了相同的寄存器。 ⇒ 进行正确的编写。
#pragma DMAC, unknown register name used （#pragma DMAC，使用了未知的寄存器名称）（仅限于 NC308）	• 在 #pragma DMAC 声明中使用了未知的寄存器。 ⇒ 进行正确的编写。
#pragma JSRA illegal location ,ignored （#pragma JSRA 的位置不正确，将被忽略）	• 不要将 #pragma JSRA 放置在函数范围内。 ⇒ 在函数之外编写 #pragma JSRA。
#pragma JSRW illegal location ,ignored（#pragma JSRW 的位置不正确，将被忽略）	• 不要将 #pragma JSRW 放置在函数范围内。 ⇒ 在函数之外编写 #pragma JSRA。
#pragma PARAMETER function's address used （使用了 #pragma PARAMETER 函数的地址）	• #pragma PARAMETER 所指定的函数地址被赋值到指针变量。 ⇒ 和不赋值一样，请正确编写。
#pragma control for function duplicate, ignored （函数的 #pragma 控制重复，将被忽略）	• 在 #pragma 中为相同函数指定了两个或多个 INTERRUPT、TASK、 HANDLER、 CYCHANDLER 或 ALMHANDLER。 ⇒ 确保仅指定一个 INTERRUPT、 TASK、 HANDLER、CYCHANDLER 或 ALMHANDLER。
#pragma unknown switch, ignored （未知的 #pragma 切换，将被忽略）	• 对 #pragma 指定了无效的切换。 #pragma 声明将被忽略。 ⇒ 正确的编写切换。
'auto'is illegal storage class （‘auto’是不正确的存储类）	• 使用了不正确的存储类。 ⇒ 指定正确的存储类。
'register'is illegal storage class （‘register’是不正确的存储类）	• 使用了不正确的存储类。 ⇒ 指定正确的存储类。
argument is define by 'typedef', 'typedef' ignored （参数由 ‘typedef’进行定义，‘typedef’将被忽略）	• 参数声明中使用了说明符 typedef。说明符 typedef 将被忽略。 ⇒ 删除 typedef。
assign far pointer to near pointer, bank value ignored （将 far 指针赋值到 near 指针，存储体的值将被忽略）	• 当把 far 指针替代为 near 指针时，存储体的地址将失效。 ⇒ 检查数据类型是 near 或 far。
assignment from const pointer to non-const pointer （从 const 指针赋值到 non-const 指针）	• 从 const 指针赋值到 non-const 指针，将使 const 的属性丢失。 ⇒ 检查语句的描述。如果描述正确，则忽略这项警告。

附表 F.21　　ccom30 警告信息 (3)

警告信息	描述和解决方法
assignment from volatile pointer to non-volatile pointer （从 volatile 指针赋值到 non-volatile 指针）	• 从 volatile 指针赋值到 non-volatile 指针，将使 volatile 的属性丢失。 ⇒ 检查语句的描述。如果描述正确，则忽略这项警告。
assignment in comparison statement （比较语句中的赋值）	• 在比较语句中放置了赋值表达式。 ⇒ “==”和 ‘=’可能被混淆。请进行检查。
block level extern variable initialize forbid, ignored （禁止块级的外部变量初始化，将被忽略）	• 初始化表达式被编写在函数的外部变量声明中。 ⇒ 删除初始化表达式或者更改存储类。
can't get address from register storage class variable （无法从寄存器存储类变量获得地址）	• 为存储类寄存器的变量编写了 & 运算符。 ⇒ 不要使用 & 运算符来描述存储类寄存器的变量。
can't get size of bitfield （无法获得位字段大小）	• 为 sizeof 运算符的操作数使用了位字段。 ⇒ 正确编写操作数。
can't get size of function （无法获得函数大小）	• 为 sizeof 运算符的操作数使用了函数名称。 ⇒ 正确编写操作数。
can't get size of function, unit size 1 assumed （无法获得函数大小，假设单位大小为 1）	• 函数的指针是增加 （++）或减少 （--）。将假设增加或减少值为 1 来继续处理。 ⇒ 不要增加 （++）或减少 （--）函数的指针。
char array initialized by wchar_t string （由 wchar_t string 初始化的 char 数组）	• char 类型的数组通过 wchar_t 类型进行了初始化。 ⇒ 确保初始化表达式的类型匹配。
case value is out of range （case 值超出范围）	• case 值超出了 switch 参数范围。 ⇒ 进行正确的指定。
character buffer overflow （字符缓冲器溢出）	• 字符串的大小超过了 512 个字符。 ⇒ 字符串使用的字符数不要超过 512 个。
character constant too long （字符常数太长）	• 字符常数中有太多的字符 （单引号内的字符）。 ⇒ 进行正确的编写。
constant variable assignment （常数变量赋值）	• 在此赋值语句中， const 修饰符指定的变量被替代。 ⇒ 检查要替代的声明部分。
cyclic or alarm handler function has argument （循环或警报处理程序函数存在参数）	• #pragma CYCHANDLER 或 ALMHANDLER 所指定的函数使用了参数。 ⇒ 函数无法使用参数。删除参数。
enumerator value overflow size of unsigned char （枚举的值溢出了 unsigned char 的大小）	• 枚举的值超过了 255。 ⇒ 不要对枚举使用超过 255 的值，否则不要指定启动函数 -fchar_enumerator。
enumerator value overflow size of unsigned int （枚举的值溢出了 unsigned int 的大小）	• 枚举的值超过了 65535。 ⇒ 不要使用超过 65535 的值来描述枚举。
enum's bitfield （enum 的位字段）	• 枚举被用作了位字段成员。 ⇒ 使用其它类型的成员。
external variable initialized,change to public （外部变量已被初始化，更改为公用）	• 为外部声明的变量指定了初始化表达式。 extern 将被忽略。 ⇒ 删除 extern。

附表 F.22　　ccom30 警告信息 (4)

警告信息	描述和解决方法
far pointer (implicitly) casted by near pointer（far 指针被隐含地转型为 near 指针）	• far 指针被转换为 near 指针。 ⇒ 检查数据类型是 near 或 far。
function must be far （函数必须为 far）	• 使用 near 类型声明了函数。 ⇒ 进行正确的编写。
function function name has no-used argument (variable-name) （函数函数名称具有未使用的参数 （变量名称））	• 在参数中对函数声明的变量未被使用。 ⇒ 检查所使用的变量。
handler function called （已调用处理程序函数）	• 调用了 #pragma HANDLER 指定的函数。 ⇒ 谨慎处理以避免调用处理程序。
handler function can't return value （处理程序函数无法返回值）	• #pragma HANDLER 所指定的函数使用了返回值。 ⇒ #pragma HANDLER 所指定的函数不可使用返回值。请将返回值删除。
handler function has argument （处理程序函数存在参数）	• #pragma HANDLER 所指定的函数使用了参数。 ⇒ #pragma HANDLER 所指定的函数不可使用参数。请将参数删除。
hex character is out of range （十六进制字符超出范围）	• 字符常数中的十六进制字符超出了长度。同时，\ 之后包含了一些不是十六进制表达的字符。 ⇒ 减少十六进制字符的长度。
identifier (member-name) is duplicated, this declare ignored （标识符 （成员名称）重复，这项声明将被忽略）	• 成员名称被定义了两次或更多次。这项声明将被忽略。 ⇒ 确保成员名称只声明一次。
identifier (variable-name) is duplicated （标识符 （变量名称）重复）	• 变量名称被定义了两次或更多次。这项声明将被忽略。 ⇒ 确保变量名称只声明一次。
identifier (variable-name) is shadowed （标识符 （变量名称) 被遮住）	• 使用了与参数的声明名称相同的 auto 变量。 ⇒ 使用任何未用于参数的名称。
illegal storage class for argument, 'extern' ignore （不正确的参数存储类，‘extern’将被忽略）	• 在函数定义的参数列表中使用了无效的存储类。 ⇒ 指定正确的存储类。
incomplete array access （不完整的数组存取）	• 存取了不完整的多维数组。 ⇒ 指定多维数组的大小。
incompatible pointer types （不兼容的指针类型）	• 指针指向的目标类型不正确。 ⇒ 检查指针类型。
incomplete return type （不完整的返回类型）	• 尝试引用类型不完整的返回变量。 ⇒ 检查返回变量。
incomplete struct member （不完整的 struct 成员）	• 尝试引用不完整的 struct 成员。 ⇒ 先定义完整的 struct 或 union。
init elements overflow,ignored （初始化变量元素时发生溢出，将被忽略）	• 初始化表达式超过了所要初始化变量的大小。 ⇒ 确保初始化表达式的数量不超过所要初始化变量的大小。
inline function is called as normal function before, change to static function （直接插入函数之前曾作为普通函数调用，更改为 static 函数）	• 直接插入存储类中声明的函数曾作为普通函数调用。 ⇒ 总是确保在使用直接插入函数前先进行定义。

附表 F.23　　ccom30 警告信息 (5)

警告信息	描述和解决方法
integer constant is out of range （整数常数超出范围）	• 整数常数的值超过了 unsigned long 可以表达的值。 ⇒ 使用 unsigned long 可以表达的值来描述常数。
interrupt function called （已调用中断函数）	• 调用了 #pragma INTERRUPT 指定的函数。 ⇒ 谨慎处理以避免调用中断处理函数。
interrupt function can't return value （中断函数无法返回值）	• #pragma INTERRUPT 指定的中断处理函数使用了返回值。 ⇒ 返回值不可在中断函数中使用。请将返回值删除。
interrupt function has argument （中断函数存在参数）	• #pragma INTERRUPT 指定的中断处理函数使用了参数。 ⇒ 参数不可在中断函数中使用。请将参数删除。
invalid #pragma EQU （无效的 #pragma EQU）	• #pragma EQU 的描述存在错误。这一行将被忽略。 ⇒ 正确编写描述。
invalid #pragma SECTION, unknown section base name （无效的 #pragma SECTION，未知的段基本名称）	• #pragma SECTION 中的段名称存在错误。可指定的段名称是 data、bss、program、rom、interrupt 及 bas。这一行将被忽略。 ⇒ 正确编写描述。
invalid #pragma operand, ignored （无效的 #pragma 操作数，将被忽略）	• #pragma 的操作数存在错误。这一行将被忽略。 ⇒ 正确编写描述。
invalid function argument （无效的函数参数）	• 函数参数未正确编写。 ⇒ 正确编写函数参数。
invalid return type （无效的返回类型）	• 返回语句的表达式与函数类型不匹配。 ⇒ 确保返回值与函数类型匹配，或者函数类型与返回值匹配。
invalid storage class for function, change to extern （无效的函数存储类，更改为 extern）	• 函数声明中使用了无效的存储类。在处理时，它将作为 extern 处理。 ⇒ 将存储类更改为 extern。
Kanji in #pragma ADDRESS （#pragma ADDRESS 中的日文汉字）	• #pragma ADDRESS 的行存在日文汉字代码。这一行将被忽略。 ⇒ 不要在此声明中使用日文汉字代码。
Kanji in #pragma BITADDRESS （#pragma BITADDRESS 中的日文汉字）	• #pragma BITADDRESS 的行存在日文汉字代码。这一行将被忽略。 ⇒ 不要在此声明中使用日文汉字代码。
keyword (keyword) are reserved for future （关键字 （关键字）保留以供将来使用）	• 使用了保留的关键字。 ⇒ 将它更改为另一名称。
large type was implicitly cast to small type （大类型被隐含地转型为小类型）	• 从大类型赋值到较小的类型，将使值的高位字节 （字）丢失。 ⇒ 检查类型。如果描述正确，则忽略这项警告。
mismatch prototyped parameter type （已声明原型的参数类型不匹配）	• 参数类型不是在原型声明中声明的类型。 ⇒ 检查参数类型。

附表 F.24　　ccom30 警告信息 (6)

警告信息	描述和解决方法
meaningless statements deleted in optimize phase （在优化阶段删除了无意义的语句）	• 在优化过程中删除了无意义的语句。 ⇒ 删除无意义的语句。
meaningless statement （无意义的语句）	• 语句的结尾是 “==”。 ⇒ “=”和 ‘==’可能被混淆。请进行检查。
mismatch function pointer assignment （不匹配的函数指针赋值）	• 具有寄存器参数的函数地址，被替代为不具有寄存器参数的函数指针 （即，未声明原型的函数）。 ⇒ 将函数的指针变量声明更改为原型声明。
multi-character character constant （多字符的字符常数）	• 使用了由两个或更多个字符组成的字符常数。 ⇒ 在需要两个或更多个字符时使用宽字符 （L ‘xx’）。
near/far is conflict beyond over typedef（near/far 在 typedef 之上冲突）	• 通过指定 near/far 定义的类型，在引用时通过指定 near/far 而再次进行了定义。 ⇒ 正确编写类型说明符。
No hex digit （无十六进制数字）	• 十六进制常数包含了一些无法在十六进制记数法中使用的字符。 ⇒ 使用 0 到 9 的数字和 A 到 F 及 a 到 f 的字母来描述十六进制常数。
No initialized of variable's name （变量名称未初始化）	• 所使用的寄存器变量可能未进行初始化。 ⇒ 确保寄存器变量被赋予了适当的值。
No storage class & data type in declare, global storage class & int type assumed （声明中无存储类和数据类型，将被假设为全局存储类和 int 类型）	• 变量未使用存储类和类型说明符进行声明。在处理时，它将作为 int 处理。 ⇒ 编写存储类和类型说明符。
non-initialized variable "variable name" is used （使用了未初始化的变量 “变量名称”）	• 可能引用了未初始化的变量。 ⇒ 检查语句的描述。这项警告会在函数的最后一行发生。在这种情况下，检查函数中的 auto 变量等的描述。如果描述正确，则忽略这项警告。
non-prototyped function used （使用了未声明原型的函数）	• 调用了未声明原型的函数。仅在指定了 -Wnon_prototype 选项时才输出此信息。 ⇒ 编写原型声明。或删除选项 “- Wnon_prototype”。
non-prototyped function declared （声明了未声明原型的函数）	• 找不到所定义函数的原型声明。 （只在指定了 -Wnon_prototype 选项时显示。） ⇒ 编写一个原型声明。
octal constant is out of range （八进制常数超出范围）	• 八进制常数包含了一些无法在八进制记数法中使用的字符。 ⇒ 使用 0 到 7 的数字来描述八进制常数。
octal_character is out of range （八进制字符超出范围）	• 八进制常数包含了一些无法在八进制记数法中使用的字符。 ⇒ 使用 0 到 7 的数字来描述八进制常数。
overflow in floating value converting to integer （浮点值转换为整数时溢出）	• 无法以整数类型存储的庞大浮点数被赋值为整数类型。 ⇒ 重新检查赋值表达式。

附表 F.25　　ccom30 警告信息 (7)

警告信息	描述和解决方法
old style function declaration （旧式的函数声明）	• 编写函数定义时使用了 ANSI （ISO） C 之前的格式。 ⇒ 以 ANSI （ISO）格式来编写函数定义。
prototype function is defined as non-prototype function before （原型函数曾被定义为非原型函数）	• 未声明原型的函数被重新定义为声明了原型的函数。 ⇒ 统一声明函数类型的方式。
redefined type （重新定义了类型）	• 重新定义了 typedef。 ⇒ 请检查 typedef。
redefined type name of (qualify) （重新定义了（qualify）的类型名称）	• 相同的标识符在 typedef 中被定义了两次或更多次。 ⇒ 正确编写标识符。
register parameter function used before as stack parameter function （寄存器参数函数曾被用作堆栈参数函数）	• 寄存器参数函数曾被用作堆栈参数函数。 ⇒ 使用函数之前先编写原型声明。
RESTRICT qualifier can set only pointer type （RESTRICT 修饰符只能设置指针类型）	• RESTRICT 修饰符在指针外进行了声明。 ⇒ 仅在指针内声明它。
section name 'interrupt' no more used （段名称‘interrupt’已不再使用）	• #pragma SECTION 所指定的段名使用了 ‘interrupt’。 ⇒ 不可使用 ‘interrupt’的段名称。将它更改为另一名称。
size of incomplete type （不完整类型的大小）	• 在 size of 运算符的操作数中使用了未定义的结构或联合。 ⇒ 先定义结构或联合。
	• 定义为 size of 运算符的操作数数组所具有的元素数量未知。 ⇒ 先定义结构或联合。
size of incomplete array type （不完整数组类型的大小）	• 尝试查找了大小未知的数组的 size of。这是无效的大小。 ⇒ 指定数组的大小。
size of void （void 的大小）	• 尝试查找了 void 的大小。这是无效的大小。 ⇒ 找不到 void 的大小。
standard library "function-name()" need "include-file name" （标准程序库 “函数名称 ()”需要“包含文件名称”）	• 此标准程序库函数的使用未包括标题文件。 ⇒ 确保包括了标题文件。
static variable in inline function （直接插入函数中的 static 变量）	• 在直接插入存储类所声明的函数中声明了 static 数据。 ⇒ 不要在直接插入函数中声明 static 数据。
string size bigger than array size （字符串的大小大于数组的大小）	• 初始化表达式的大小大于所要初始化变量的大小。 ⇒ 确保初始化表达式的大小等于或小于变量。
string terminator not added （未添加字符串终止符）	• 由于所要初始化的变量和初始化表达式的大小相同，因此无法将‘\0’附加到字符串。 ⇒ 增加数组的元素数。
struct (or union) member's address can't has no near far information （struct （或 union）成员的地址不可具有 near、 far 信息）	• near 或 far 被用作 struct （或 union）成员 （变量）的位置安排信息。 ⇒ 不要为成员指定 near 和 far。

附表 F.26　　ccom30 警告信息 (8)

警告信息	描述和解决方法
task function called （已调用任务函数）	• 调用了 #pragma TASK 所指定的函数。 ⇒ 谨慎处理以避免调用任务函数。
task function can't return value （任务函数无法返回值）	• #pragma TASK 所指定的函数使用了返回值。 ⇒ #pragma TASK 所指定的函数不可使用返回值。请将返回值删除。
task function has invalid argument （任务函数存在无效的参数）	• #pragma TASK 所指定的函数使用了参数。 ⇒ 任何 #pragma TASK 所指定的函数均不可使用参数。请将参数删除。
this comparison is always false （此比较结果总是为 “假”）	• 所进行比较的结果总是为 “假”。 ⇒ 检查条件表达式。
this comparison is always true （此比较结果总是为 “真”）	• 所进行比较的结果总是为 “真”。 ⇒ 检查条件表达式。
this feature not supported now, ignored （现在不支持这项功能，将被忽略）	• 这是一项语法错误。此语法已保留以供将来的扩展使用，因此请不要使用此语法。 ⇒ 正确编写描述。
this function used before with non-default argument （此函数曾和非默认参数一起使用）	• 此函数曾声明为具有默认参数的函数进行使用。 ⇒ 在使用函数之前先声明默认参数。
this interrupt function is called as normal function before （此中断函数曾作为普通函数调用）	• 在 #pragma INTERRUPT 中声明了之前曾使用的函数。 ⇒ 无法调用中断函数。请检查 #pragma 的内容。
too big octal character （八进制的字符太大）	• 字符串中的字符常数或八进制常数超过了限制值 （以十进制表示的 255）。 ⇒ 不要使用大于 255 的值来描述常数。
too few parameters （太少参数）	• 与原型声明中所声明的参数数量相比，参数的数量不足。 ⇒ 检查参数的数量。
too many parameters （太多参数）	• 与原型声明中所声明的参数数量相比，参数的数量太多。 ⇒ 检查参数的数量。
unknown #pragma STRUCT xxx （未知的 #pragma STRUCT xxx）	• 无法处理 #pragma STRUCTxxx。这一行将被忽略。 ⇒ 进行正确的编写。
Unknown debug option (-dx) （未知的调试选项 （-dx））	• 无法指定选项 -dx。 ⇒ 正确指定选项。
Unknown function option (-Wxxx) （未知的函数选项 （-Wxxx））	• 无法指定选项 -Wxxx。 ⇒ 正确指定选项。
Unknown function option (-fx) （未知的函数选项 （-fx））	• 无法指定选项 -fx。 ⇒ 正确指定选项。
Unknown function option (-gx) （未知的函数选项 （-gx））	• 无法指定选项 -gx。 ⇒ 正确指定选项。
Unknown optimize option (-mx) （未知的优化选项 （-mx））	• 无法指定选项 -mx。 ⇒ 正确指定选项。
Unknown optimize option (-Ox) （未知的优化选项 （-Ox））	• 无法指定选项 -Ox。 ⇒ 正确指定选项。
Unknown option (-x) （未知的选项 （-x））	• 无法指定选项 -x。 ⇒ 正确指定选项。

附表 F.27　　ccom30 警告信息 (9)

警告信息	描述和解决方法
unknown pragma pragma-specification used （使用了未知的 pragma 杂注指定）	• 编写了不支持的 #pragma。 ⇒ 检查 #pragma 的内容。 * 仅在指定了 Wunknown_pragma (-WUP) 选项时才显示这项警告。
wchar_t array initialized by char string （wchar_t 数组由 char 字符串初始化）	• wchar_t 类型的初始化表达式由 char 类型的字符串进行了初始化。 ⇒ 确保初始化表达式的类型匹配。
zero divide in constant folding （常数合并中的零除数）	• 除法运算符或余数计算运算符中的除数为 0。 ⇒ 对除数使用 0 以外的任何其它值。
zero divide,ignored （零除数，将被忽略）	• 除法运算符或余数计算运算符中的除数为 0。 ⇒ 对除数使用 0 以外的任何其它值。
zero width for bitfield （位字段的零宽度）	• 位字度宽度为 0。 ⇒ 编写等于或大于 1 的位字段。
no const in previous declaretion （之前的声明中无 const）	• 不具有 const 修饰的函数或变量声明在实体定义侧进行了 const 修饰。 ⇒ 确保函数或变量声明与实体定义侧的 const 修饰匹配。
Code generation for static functions (xxx) can be suppressed by using -ferase_static_function(-fESF) option （static 函数 （xxx）的代码生成可以通过使用 -ferase_static_function(-fESF) 的选项来禁止）	• 一些 static 函数可能无法被引用。 ⇒ static 函数 （函数名称）的代码生成可以通过指定 -ferase_static_function 的选项来禁止。

附录 G　SBDATA 声明及 SPECIAL 页函数声明工具 (utl30)

下面描述如何启动 SBDATA 声明及 SPECIAL 页函数声明工具 (utl30)，及启动选项的运行方式。

附录 G.1　utl30 简介

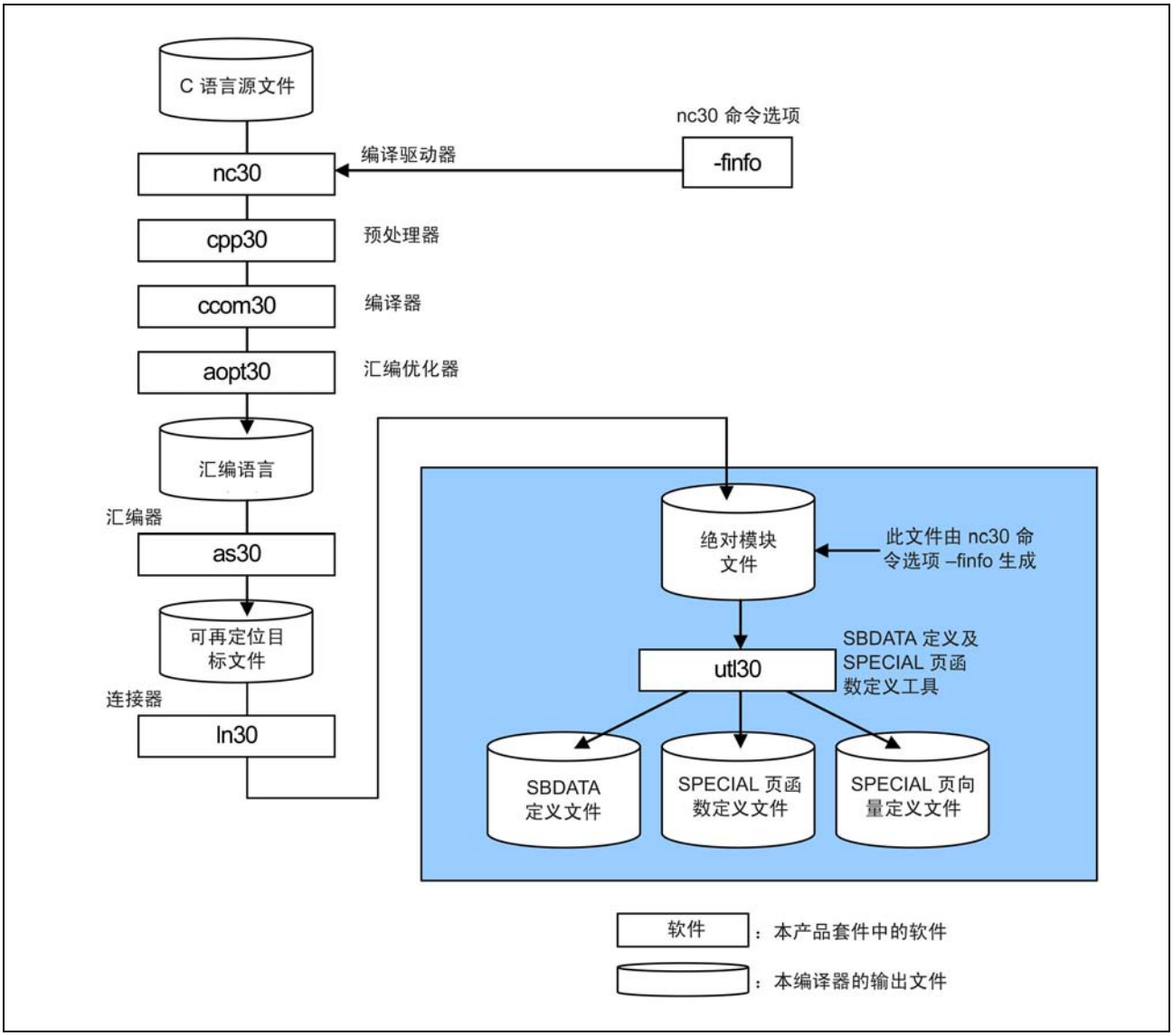
附录 G.1.1　utl30 处理的简介

SBDATA 声明及 SPECIAL 页函数声明工具 utl30 的用途是处理绝对模块文件 （扩展名为 .x30）。

utl30 会生成一个包含 SBDATA 声明 （位于 SB 区内，以最常用的 “#pragma SBDATA”作为起始）的文件，及一个包含 SPECIAL 页函数声明 （位于 SPECIAL 页区内，以最常用的 “#pragma SPECIAL”作为起始）的文件。

若要使用 utl30，在编译时指定编译驱动器启动选项 -finfo，以便生成绝对模块文件 (.x30)。

附图 G.1 说明 NC30 的处理流程。



附图 G.1　　NC30 处理流程

附录 G.2 启动 utl30

附录 G.2.1 utl30 命令行格式

要启动 utl30，您必须指定所需的信息和参数。

```
% utl30Δ[ 命令行选项 ]. < 绝对文件名 >

%    :  提示符
< >  :  强制性项目
[ ]  :  可选项目
Δ    :  空格
使用空格分隔多个命令行选项。
```

附图 G.2 utl30 命令行格式

在开始使用 utl30 前，必须先指定下列的编译器启动选项，以便生成绝对模块文件 （扩展名 .x30）。

- -finfo 选项用于输出检查器信息
- -g 选项用于输出调试信息

也必须指定以下 utl30 选项：

- -o 选项用于输出信息 （SBDATA 声明或 SPECIAL 页函数声明）

（默认情况下，信息会被输出到标准输出器件。）

• 输出绝对模块文件

```
%nc30 ncrt0.a30 -finfo sample.c<RET>
M16C/60, 30, 20, 10, Tiny, R8C/Tiny Series Compiler V.x.xx Release xx
Copyright(C) xxxx(xxxx). Renesas Technology Corp.
and Renesas Solutions Corp., All rights reserved.
ncrt0.a30
sample.c

%
```

• 输出 SBDATA 声明

```
%utl30 -sb30 ncrt0.x30 -o sample<RET>
M16C/60 UTILITY UTL30 for M16C/60 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%
```

• 输出 SPECIAL 页函数声明

```
%utl30 -sp30 ncrt0.x30 -o sample <RET>
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
```

<RET> : 表示按下回车键。

附图 G.3　　utl30 命令行的范例

附录 G.2.2　　选择输出信息

若要为 utl30 选择 “SBDATA declaration”（SBDATA 声明）或 “SPECIAL page function declaration”（SPECIAL 页函数声明）输出，必须指定下面所述的选项。若未指定任何选项， utl30 将发生错误。

1. 输出 SBDATA 声明
   - 选项 “-sb30”
2. 输出 SPECIAL 页函数声明
   - 选项 “-sp30”

附图 G.3 显示 sbutl 的命令行选项。

附录 G.2.3 utl30 的命令行选项

启动 utl30 必须使用以下信息 （输入参数）。附表 G.1 显示 utl30 的命令行选项。

附表 G.1 utl30 的命令行选项

选项	缩写	描述
-all	无	[当与 -sb30 选项同时使用时] 由于使用的频率低，对不在 SB 区内的变量， SBDATA 声明会以注释的方式输出。 [当与 -sp30 选项同时使用时] 由于使用的频率低，对不在 SPECIAL 页区域内的函数， SPECIAL 声明会以注释的方式输出。
-fsection	无	#pragma SECTION 所指定的变量及函数也包括在处理的范围内。
-fover_write	-fOW	强制盖写以 -o 选项指定的输出文件名。
-o	无	将 SBDATA 声明或 SPECIAL 页函数声明的结果输出到文件。若未指定这个选项，则结果会被输出到主机 （EWS 或个人计算机）的标准输出器件上，且无法指定任何扩展名。 若所指定的文件已存在，则结果将会直接写入到标准输出器件。
-sb30	无	-sb30 -> 输出 SBDATA 声明。 -sp30 -> 输出 SPECIAL 页函数声明。 若要使用 utl30，总是指定两个选项中的其中一个。 若未指定任何选项，则会发生错误。
-sp=< 编号 > --sp=< 编号 >,< 编号 >,... （两个或更多个编号） -sp=< 编号 >-< 编号 >	无	指定的编号不会用作 SPECIAL 页函数的编号。 与 -sb30 选项同时使用此选项。
-sp30	无	-sb30 -> 输出 SBDATA 声明。 -sp30 -> 输出 SPECIAL 页函数声明。 若要使用 utl30，总是指定两个选项中的其中一个。 若未指定任何选项，则会发生错误。
-Wstdout	无	输出警告和错误信息到主机的标准输出器件。

-all

所有变量的 SBDATA 声明或对所有函数的 SPECIAL 页函数声明的输出

功能：
- 当与 -sb30 选项同时使用时
  由于使用的频率低，对不在 SB 区内的变量， SBDATA 声明会以注释的方式输出。
- 当与 -sp30 选项同时使用时
  由于使用的频率低，对不在 SPECIAL 页区域内的函数， SPECIAL 声明会以注释的方式输出。

补充：使用这个选项可帮助找出不曾在程序执行中调用一次的函数。
不过，必须注意仅通过间接方式调用的函数，因为这些函数的被调用次数会显示为 0。

-fover_write　　-fOW

将 SBDATA 声明或 SPECIAL 函数声明输出到文件

功能：不会检查 -o 所指定的输出文件是否已存在。若文件存在，则将对它进行盖写。
此选项必须与 -o 选项一同指定。

-fsection

输出 #pragma SECTIONS 中的 SBDATA 声明和 SPECIAL 页函数声明

功能：段名称被 #pragma SECTION 更改的存储区中的变量和函数也包括在处理范围内。

注意：若使用 #pragma SECTION 以一个特定地址查找一个特定变量或函数，则请勿指定此选项，因为 SBDATA 或 SPECIAL 页声明可能使变量或函数处在未预期的不同地址上。

-o

输出已声明的 SBDATA 结果显示文件

功能：将 SBDATA 声明或 SPECIAL 页函数声明的结果输出到文件。若未指定这个选项，则结果会被输出到主机 （EWS 或个人计算机）的标准输出器件上，若所指定的文件已存在，则结果将会直接写入到标准输出器件。

-sb30

输出 SBDATA 声明

功能：输出 SBDATA 声明。此选项可以和 -sp30 同时指定。

-sp30

输出 SPECIAL 页函数声明

功能： 输出 SPECIAL 页函数声明。不指定的情况下会发生错误。

-sp= < *编号* >

指定不被用作 SPECIAL 页函数的编号

功能： 指定不被用作 SPECIAL 页函数的编号。
此选项可以和 -sp30 同时使用。

-Wstdout

标准输出的警告显示

功能： 将错误和警告信息输出到主机的标准输出 （stdout）。

附录 G.3 注意

1. 当使用 utl30 时，将无法对在汇编器中描述的文件中所声明的 .sbsym 进行计数。基于这个原因，当在汇编器中声明 “.sbsym”时，必须做出调整，以便将执行 utl30 所产生的结果放置到 SB 区。
2. 当使用 utl30 时，将无法对在汇编器中描述的文件中所声明的 SPECIAL 页函数进行计数。基于这个原因，当存在在汇编器中声明的 SPECIAL 页函数时，您必须做出调整，以便将执行 utl30 所产生的结果放置到 SPECIAL 页区。

附录 G.4 建立 SBDATA 声明和 SPECIAL 页函数声明的条件

附录 G.4.1 建立 SBDATA 声明的条件

当使用 utl30 时，只有全局变量有效。可用的变量类型如下：

- _Bool 的变量
- unsigned char 和 signed char 类型的变量
- unsigned short 和 signed short 类型的变量
- unsigned int 和 signed int 类型的变量
- unsigned long 和 signed long 类型的变量
- unsigned long long 和 signed long long 类型的变量

下面是不包括在 SBDATA 声明中的变量。

- 位于段的变量将随着 #pragma SECTION 的指定而改变
- #pragma ADDRESS 所定义的变量
- #pragma ROM 所定义的变量

当使用 utl30 时，若程序中已存在使用 #pragma SBDATA 来声明的变量，该声明将具有较高的优先级，同时将其它的变量分配到其余的 SB 区中。

附录 G.4.2 建立 SPECIAL 页函数声明的条件

由 utl30 处理的函数只限于下面所列出的外部函数。

- 不是使用 static 来声明的函数
- 被调用四次或更多次的函数

然而，必须注意，当上述函数属于下列其中一项时，它们也不会被处理：

- 位于 #pragma SECTION 所指定的段的函数
- 任何 #pragma 所定义的函数

当使用 utl30 时，若程序中已存在使用 #pragma SPECIAL 来声明的变量，该声明将具有较高的优先级，同时将其它的变量分配到其余的 SB 区中。

附录 G.5 utl30 的使用范例

附录 G.5.1 生成 SBDATA 声明文件

(a) 生成 SBDATA 声明文件

通过让 utl30 （用编译选项 finfo）处理绝对模块文件以输出 SBDATA 声明文件。
附图 G.4 显示在 utl30 中进行输入处理的范例，而附图 G.5 则显示 SBDATA 声明文件的范例。

```
% utl30 -sb30 ncrt0.x30 -osbdata<RET>

%: 提示符
ncrt0.x30 : 绝对文件名称
```

附图 G.4 utl30 命令行的范例

```
/*
* #pragma SBDATA 工具
*/
/* SBDATA 大小 [255] */
#pragma SBDATA      data3       /* size = (4)/ref = [ 2] */
#pragma SBDATA      data2       /* size = (1)/ref = [ 1] */
#pragma SBDATA      data1       /* size = (2)/ref = [ 1] */
                                  (1)              (2)
/*
* 文件结束
*/

(1)Size=() 是数据大小
(2)ref =() 是变量的存取计数
```

附图 G.5 SBDATA 声明文件 (sbdata.h)

可以将上面生成的 SBDATA 声明文件作为标题文件包含在程序中。附图 G.6 显示在 SBDATA 文件中进行设置的范例。

```
#include  "sbdata.h"

void      func(void)
{
          （已省略）
              :
```

附图 G.6 在 SBDATA 中进行设置的范例

(b) 对在汇编器中做出 SB 声明的程序进行调整的范例

若因为汇编器例程中的 .sbsym 声明而使用了 SB 区，必须调整 utl30 所生成的文件。

[汇编器例程]

```
        .sbsym      _sym
            :
        （已省略）
            :
        .glb        _sym
_sym:
        .blkb       2
```

[utl30 所生成的文件]

```
/*
 * #pragma SBDATA 工具
 */
/* SBDATA 大小 [255] */
#pragma SBDATA      data3           /* size = (4) / ref = [ 2] */
#pragma SBDATA      data2           /* size = (1) / ref = [ 1] */
            :
        （已省略）
            :
#pragma SBDATA      data1           /* size = (2) / ref = [ 1] */
/*
 * 文件结束
 */
```

由于在汇编器例程中将 2 字节的数据进行了 SB 声明，因此需要从 utl30 所生成的文件减去 2 字节的 SBDATA 声明。

范例 :

```
            :
        （已省略）
            :
//#pragma SBDATA    data1           /* size = (2) / ref = [ 1] */
/* 变为注释 */
```

附图 G.7 调整由 utl30 所生成文件的范例

附录 G.5.2　　生成 SPECIAL 页函数声明文件

(a)　生成 SPECIAL 页函数声明文件

通过由 utl30 （SBDATA 声明及 SPECIAL 页函数声明工具）处理绝对模块文件 （在编译时使用 -finfo 选项生成），可输出 SPECIAL 页函数声明及 SPECIAL 页向量定义文件。

附图 G.8 显示在 utl30 中的输入范例。附图 G.9 显示一个 SPECIAL 页函数声明文件的范例。附图 G.10 显示一个 SPECIAL 页向量定义文件的范例。

```
% utl30  -sp30  ncrt0.x30  -o special<RET>

% : 提示符
ncrt0.x30 : 绝对文件名称
```

附图 G.8　　utl30 命令行的范例

```
/*
 * #pragma SPECIAL PAGE 工具
 */
/* SBDATA 大小 [255] */
#pragma  SPECIAL    255          func1      /* size = (100), ref = [ 10] */
#pragma  SPECIAL    254          func2      /* size = (100), ref = [ 7] */
#pragma  SPECIA     253          func3      /* size = (100), ref = [ 5] */
                                            (1)                    (2)
/*
 * 文件结束
 */


(1) 表示函数大小。
(2) 表示函数的引用频率。
```

附图 G.9　　SPECIAL 页函数声明文件 (special.h)

```
;
; #pragma SPECIAL PAGE 工具
;
; special 页定义
;
SPECIAL.macroNUM
        .org    0FFFFEH-(NUM*2)
        .glb    __SPECIAL_@NUM
        .word   __SPECIAL_@NUM  & 0FFFFH
        .endm
        SPECIAL255
        SPECIAL254
        SPECIAL253
;
; 文件结束
;
```

附图 G.10　　SPECIAL 页向量声明文件 (special.inc)

可以将上面生成的 SPECIAL 页函数声明文件作为标题文件包含在程序中。附图 G.11 显示在 SPECIAL 页函数声明文件中进行设置的范例。

```
#include  "special.h"

void      func(void)
{
          （已省略）
              :
```

附图 G.11　在 SPECIAL 页函数文件中进行设置的范例

在启动时包含 SPECIAL 页向量定义文件，作为所要包含的文件。附图 G.12 显示设置一个 SPECIAL 页向量定义文件的范例。

```
              :
          （已省略）
              :
          .section      vector
          .include      "special.inc"
              :
          （已省略）
              :
```

附图 G.12　在 SPECIAL 页函数文件中进行 sect30.inc 的设置的范例

附录 G.6 utl30 错误信息

附录 G.6.1 错误信息

附表 G.2 列出了 utl30 计算工具的错误信息及它们的解决方法。

附表 G.2 sbutl 错误信息

错误信息	错误内容和更正方法
ignore option '?' （忽略选项 ‘?’）	• 您指定了一个不可以在 utl30 中使用的选项。 ⇒ 指定适当的选项。
Illegal file extension'.XXX' （不正确的文件扩展名 ‘.XXX’）	• 输入文件具有不正确的文件扩展名。 ⇒ 指定适当的文件。
No input "x30" file specified （未指定任何输入 “x30” 文件）	• 没有映射文件。 ⇒ 指定映射文件。
cannot open "x30" file 'file-name' （无法打开 “x30” 文件 ‘文件名’）	• 未找到映射文件。 ⇒ 指定正确的输入映射文件。
cannot close file 'file-name' （无法关闭文件 ‘文件名’）	• 无法关闭输入文件。 ⇒ 指定正确的输入文件名称。
cannot open output file 'file-name' （无法打开输出文件 ‘文件名’）	• 无法关闭输出文件。 ⇒ 指定正确的输出文件名称。
not enough memory （存储器不足）	• 扩展存储器不足。 ⇒ 增加扩展存储器。
since 'file-name' file exist, it makes a standard output （由于 ‘文件名’ 文件已存在，它将进行标准输出）	• 使用 -o 指定的 ‘file-name’ 已存在。 ⇒ 检查输出文件名称。 通过将 -fover_write 与选项同时指定，可盖写文件。

附录 G.6.2 警告信息

附表 G.3 列出了 sbutl 工具的警告信息及它们的解决方法。

附表 G.3 sbutl 警告信息

警告信息	警告内容和更正方法
conflict declare of 'variable （变量的声明冲突）	• 此处显示的变量在多个文件中使用不同的存储类、类型等进行了声明。 ⇒ 检查这个变量的声明方式。
conflict declare of 'function （函数的声明冲突）	• 此处显示的函数在多个文件中使用不同的存储类、类型等进行了声明。 ⇒ 检查这个函数的声明方式。

附录 H　使用 Call Walker 的 gensni 或 .sni 文件建立工具

在使用 HEW 的 Call Walker 或堆栈分析工具前，必须有 .sni 文件作为其输入文件。

使用 Call Walker 的 gensni 或 .sni 文件建立工具从绝对模块文件建立这些 .sni 文件。

附录 H.1　启动 Call Walker

选择注册到 HEW 的 “Call Walker”，或从 HEW 的 Tools （工具）菜单选择 Call Walker，以将它启动。

启动 Call Walker 后，从 File （文件）菜单选择 Import Stack File （导入堆栈文件），并选择一个 .sni 文件作为 Call Walker 的输入文件。

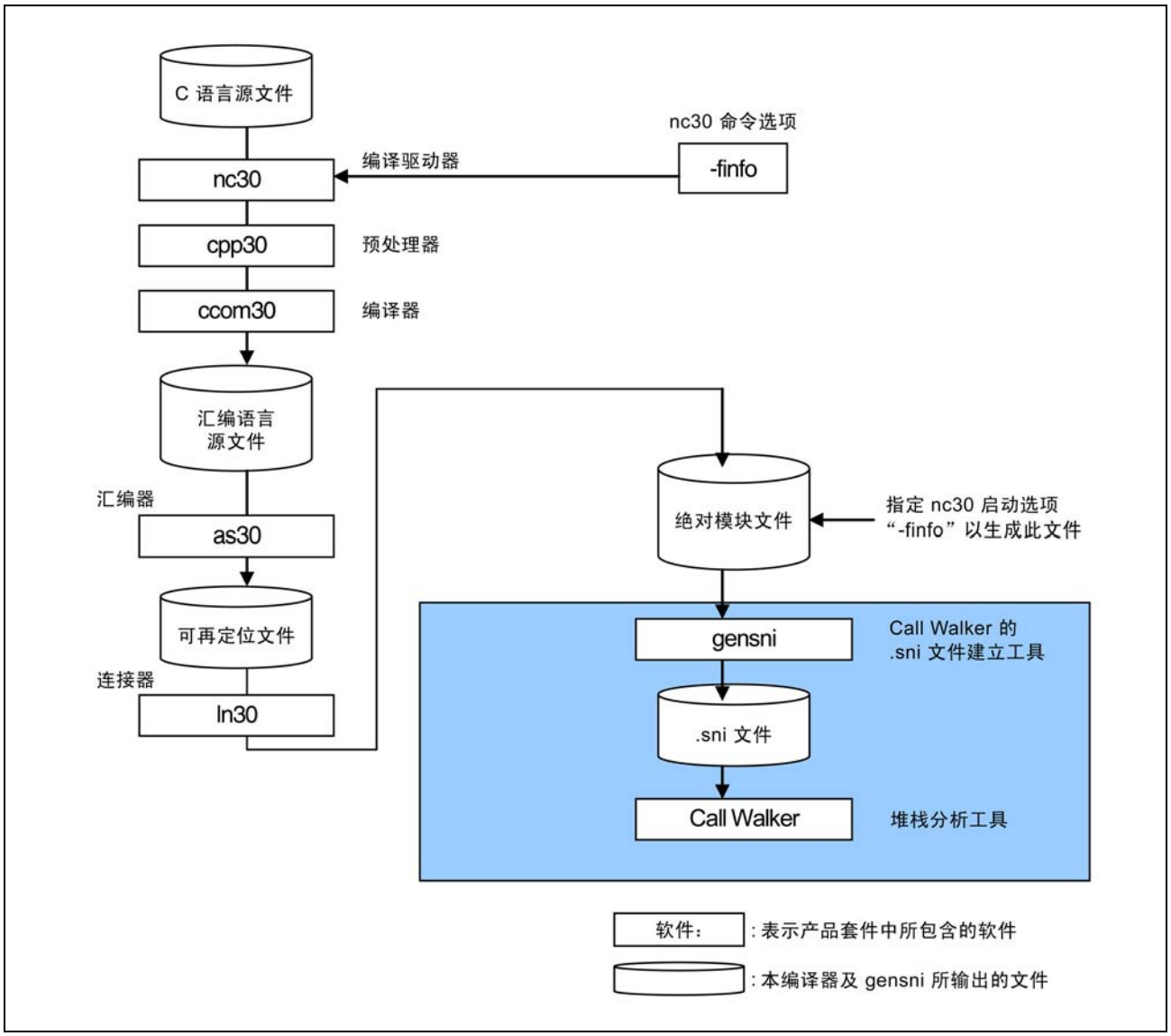
附录 H.2　gensni 概述

附录 H.2.1　gensni 处理概述

gensni 是为 Call Walker 建立 .sni 文件的工具。

gensni 通过处理绝对模块文件 （扩展名 .x30）来生成一个 .sni 文件。在使用 gensni 前，必须存在一个能够使用的绝对模块文件 （扩展名 .x30）。在进行编译时指定 “-finfo”、“-g”编译选项以生成该文件。

NC30 的处理流程在附图 H.1 中显示。



附图 H.1　　NC30 的处理流程

附录 H.3　　启动 gensni

若从 HEW 启动 Call Walker，gensni 将自动执行。但是，若不是从 HEW 启动 Call Walker，gensni 将不会自动执行。在此情况下，将需要从 Windows 的命令提示符启动 gensni。

附录 H.3.1　　输入格式

若要启动 gensni，根据下面所示的输入格式来指定一个输入文件名称和启动选项。

```
% gensniΔ[ 命令选项 ]Δ 绝对模块文件 （扩展名 .x30）

%    ：  表示提示符
< >  ：  表示必要项目。
[ ]  ：  表示在必要时编写的项目。
Δ    ：  表示空格。
当编写多个启动选项时，使用空格来将它们分隔开。
```

附图 H.2　　gensni 命令输入格式

若要使用 gensni，在本编译器的启动选项中同时指定下列两项

- 检查器信息输出 .....-finfo 选项
- 调试信息输出 .....-g 选项

以生成绝对模块文件 （扩展名 “.x30”）。

输入范例如下所示。在此处的输入范例中，为 gensni 指定了下列选项。

- 将信息输出到一个指定的文件 .....-o 选项

（默认情况下，信息会输出到一个以输入文件命名，但文件扩展名从 “.x30” 更改为 “.sni” 的文件中。）

```
生成绝对模块文件：

% nc30 -g -fansi ncrt0.a30 sample.c <RET>
M16C/60,30,20,10,Tiny,R8C/Tiny Series Compiler V.X.XX Release XX
Copyright(C) XXXX(XXXX,XXXX,XXXX,XXXX). Renesas Technology Corp.
and Renesas Solutions Corp., All rights reserved.

ncrt0.a30
sample.c

%

生成 .sni 文件

%gensni -o sample ncrt0.x30<RET>

将建立 sample.sni。

%
```

附图 H.3　　gensni 命令输入范例

附录 H.3.2 选项参考

gensni 的启动选项在附表 H.1 中列出。

附表 H.1 gensni 命令选项

选项	缩写	功能
-o 文件名	无	指定一个 .sni 文件名。 • 若未指定这个选项， .sni 文件将按照输入文件来命名，但文件扩展名更改为 “.sni”。 • 若为 .sni 文件名指定了扩展名，则该指定的扩展名将更改为 “.sni”。若未指定扩展名，则将采用 “.sni” 的扩展名。
-V	无	显示 gensni 的启动信息，并在不执行任何操作的情况下终止处理。 不生成任何 .sni 文件。

-o 文件

指定一个 .sni 文件名

功能：
- 若未指定这个选项， .sni 文件将按照输入文件来命名，但文件扩展名更改为 “.sni”。
- 若未指定扩展名，则将采用 “.sni” 的扩展名。

描述：使用此选项，可在必要时更改 .sni 文件名。
也可更改扩展名。

-V

在显示 gensni 的启动信息后终止处理

功能：显示 gensni 的启动信息，并在不执行任何操作的情况下终止处理。
- 不生成任何 .sni 文件。

M16C/60、 M16C/30、 M16C/20、 M16C/10、
M16C/Tiny、 R8C/Tiny 系列
C 编译器套件 V.5.43 - C 编译器用户手册

Publication Date:	Rev1.00, Sep. 27, 2007
Published by:	Sales Strategic Planning Div. Renesas Technology Corp.
Edited by:	Customer Support Department Global Strategic Communication Div. Renesas Solutions Corp.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

RENESAS SALES OFFICES

http://www.renesas.com

Refer to "**http://www.renesas.com/en/network**" for the latest and detailed information.

Renesas Technology America, Inc.
450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

Renesas Technology Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K.
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology (Shanghai) Co., Ltd.
Unit 204, 205, AZIACenter, No.1233 Lujiazui Ring Rd, Pudong District, Shanghai, China 200120
Tel: <86> (21) 5877-1818, Fax: <86> (21) 6887-7898

Renesas Technology Hong Kong Ltd.
7th Floor, North Tower, World Finance Centre, Harbour City, 1 Canton Road, Tsimshatsui, Kowloon, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2730-6071

Renesas Technology Taiwan Co., Ltd.
10th Floor, No.99, Fushing North Road, Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

Renesas Technology Singapore Pte. Ltd.
1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

Renesas Technology Korea Co., Ltd.
Kukje Center Bldg. 18th Fl., 191, 2-ka, Hangang-ro, Yongsan-ku, Seoul 140-702, Korea
Tel: <82> (2) 796-3115, Fax: <82> (2) 796-2145

Renesas Technology Malaysia Sdn. Bhd
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No.18, Jalan Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: <603> 7955-9390, Fax: <603> 7955-9510

Colophon 6.0

M16C/60、 M16C/30、 M16C/20、 M16C/10、 M16C/Tiny、 R8C/Tiny系列
C编译器套件V.5.43-C编译器用户手册

瑞萨电子株式会社

RCJ10J0051-0100