User's Manual

NOTICE:
There are corrections on pages 159 and 179 and an addition in -Wlarge_to_small(-WLTS) on page 108 in this document.

C/C++ Compiler Package for M16C Series and R8C Family V.6.00 C/C++ Compiler User's Manual

All information contained in these materials, including products and product specifications, represents information on the product at the time of publication and is subject to change by Renesas Electronics Corporation without notice. Please review the latest information published by Renesas Electronics Corporation through various means, including the Renesas Electronics Corporation website (http://www.renesas.com).

Renesas Electronics
www.renesas.com

Rev.1.00 Jan.16, 2011

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.

Preface

NC30 is the C compiler for the Renesas M16C Series, R8C Family. NC30 converts programs written in C into assembler source files for the M16C Series, R8C Family. You can also specify compiler options for assembling and linking to generate hexadecimal files that can be written to the microcomputer.

Please be sure to read the precautions written in this manual before using NC30.

- Microsoft, MS-DOS, Windows and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries.
- Adobe and Acrobat are registered trademarks of Adobe Systems Incorporated.

All other brand and product names are trademarks, registered trademarks or service marks of their respective holders.

Terminology

The following terms are used in this manual.

Term	Meaning
NC30	Compiler system included in this compiler
nc30	Compile driver and its executable file
AS30	Assembler package included in this compiler
as30	Relocatable macro assembler and its executable file

Description of Symbols

The following symbols are used in this manual.

Symbol	Description
#	Root user prompt
%	UNIX prompt
A>	MS-Windows(TM) prompt
<RET>	Return key
< >	Mandatory item
[ ]	Optional item
Δ	Space or tab code (mandatory)
▲	Space or tab code (optional)
: (omitted) :	Indicates that part of file listing has been omitted

Additional descriptions are provided where other symbols are used.

Contents

Chapter 1 Introduction to NC30

This chapter introduces the processing of compiling performed by NC30, and provides an example of program development using NC30.

1.1 NC30 Components

NC30 is comprised of the 10 executable files listed below.

(1)	nc30	Compile driver
(2)	rcfrt	Preprocessor
(3)	ccom30	Compiler main body
(4)	aopt30	Assembler Optimizer
(5)	sbauto	SB register automatic changeover utility
(6)	as30	Assembler driver
(7)	optlnk	Optimizing linkage editor
(8)	utl30	SBDATA declaration and SPECIAL page function declaration utility
(9)	lbg30	Library generator
(10)	CallWalker	Stack display tool

1.2 NC30 Processing Flow

Figure1.1 illustrates the NC30 processing flow.

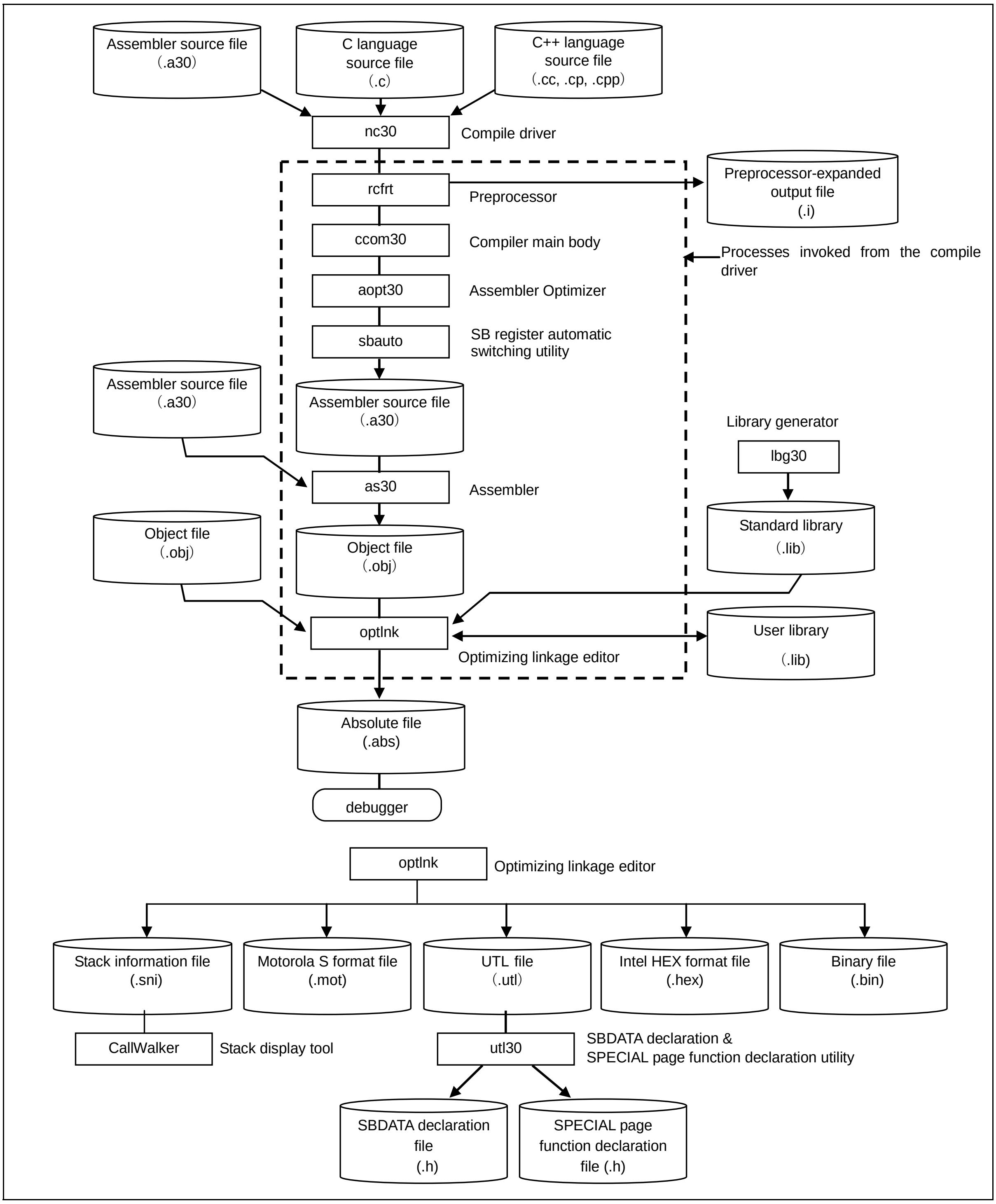


Figure1.1 NC30 Processing Flow

1.2.1 nc30

nc30 is the executable file of the compile driver.
nc30 can perform successively a series of processes from compile to link, as specified by options. Also, nc30 permits options of the relocatable macro assembler as30 and a command file of the optimizing linkage editor optlnk to be specified immediately after its startup option "-as30" and "-lnkcmd=," respectively.

1.2.2 rcfrt

rcfrt performs preprocessing.
This processing, for example, includes expanding the contents of header files and macros, as well as making decision on conditional compilation, according to the preprocessing commands that begin with #.

1.2.3 ccom30

ccom30 performs compilation.
It generates assembler source programs.

1.2.4 aopt30

aopt30 is the assembler optimizer.
It performs optimization on the assembler sources generated by ccom30.

1.2.5 sbauto

sbauto analyzes the number of times external variables are referenced in functions and thereby outputs optimum SB relative.

1.2.6 as30

The assembler (as30) loads assembler source files and assembles them to generate object files.

1.2.7 optlnk

The optimizing linkage editor (optlnk) accepts as its input the multiple object files output by the compiler and assembler and outputs load modules or library files.

1.2.8 lbg30

The standard library build tool (lbg30) is a tool to build standard library files according to user-specified options.

1.2.9 CallWalker

The stack analysis tool (CallWalker) loads the stack information files (.sni) output by the optimizing linkage editor and shows the amount of stack used.
Values indicated by Call Walker are not strictly accurate so simply use them for reference when you examine the size of the stack space. Careful evaluation is needed if you have decided the actual size of the stack space according to the information indicated by Call Walker.

1.2.10 utl30

utl30 is the executable file of a utility that generates SBDATA declarations and SPECIAL page function declarations.
utl30 loads the UTL files generated by the optimizing linkage editor and generates a file that contains an SBDATA declaration (mapped to the SB area beginning with the one that is most frequently used) and a file that contains a SPECIAL page function declaration (mapped to the SPECIAL page area beginning with the one that is most frequently used).

1.3 Notes

To use for your application the technical contents, programs, or algorithms shown in the product data, diagrams, or tables presented in this manual, please be sure to thoroughly evaluate those technical contents, programs, or algorithms as integral parts of a whole system, not just evaluating them individually as a single unit, to determine their suitability on your own responsibility. Renesas Electronics Corporation will not assume responsibility for their suitability in any particular user application.

1.3.1 Notes about Version-up of compiler

The machine language instructions (assembly language) generated by this compiler vary with the startup options specified at compile time, the contents of version upgrades, etc. Therefore, if you've changed the startup options or upgraded the compiler version, please be sure to re-evaluate the behavior of your application program.

1.3.2 Notes about the M16C's Type Dependent Part

When writing to or reading a register in the SFR area, it may sometimes be necessary to use a specific instruction. Because this specific instruction varies with each type of MCU, consult the user's manual of your MCU for details.
This compiler may generate instructions that cannot be used to write to or read from the registers in the SFR area. If an access to the SFR area is attempted as in a C program fragment in Figure 1.2, because the compiler generates instructions unusable in the SFR area, the interrupt request bit may not be determined correctly, causing an unintended behavior to occur.
When accessing registers in the SFR area in C/C++ language, write the instruction directly in the program using the asm function. In this case, make sure that the same correct instructions are generated as done by using the asm functions, regardless of the compiler's version and of whether optimizing options are used or not.

```
#pragma   ADDRESS   TA0IC   006Ch          /* M16C/60 MCU's Timer A0 interrupt control register */

struct {
          char      ILVL : 3;
          char      IR : 1;                /* An interrupt request bit */
          char      dmy : 4;
} TA0IC;

void      wait_until_IR_is_ON(void)
{
          while (TA0IC.IR == 0)            /* Waits for TA0IC.IR to become 1 */
          {
                    ;
          }
          TA0IC.IR = 0;                    /* Returns 0 to TA0IC.IR when it becomes 1 */
}
```

Figure1.2 C language description to SFR area

1.3.3 Notes on RAM Data References

To refer to the same RAM data and change its content between an interrupt handling program and an interrupted program or between tasks under the realtime OS, be sure to use volatile specification or exercise exclusive control. Also, for bit-field structures that have different member names, if their storage is reserved in the same RAM, use volatile specification or exercise exclusive control likewise

1.4 Example Program Development

A process flow of an example program development using NC30 is shown in Figure1.3. (Paragraphs (1) through (4) below correspond to the numbers (1) through (4) in Figure 1.3.)

(1) Compile the C or C++ source program (AA.c) with nc30 and assemble the output file with the assembler as30 to generate an object file (AA.obj).

(2) The startup program ncrt0.a30 and the include file sect30.inc and nc_define.inc, which contains information on the sections, are matched to the system by altering the section mapping, section size, and interrupt vector table settings.

(3) Assemble the modified startup program. As a result of this operation, an object file (ncrt0.obj) is created.

(4) Link the two object files, AA.obj and ncrt0.obj, using the optimizing linkage editor that is executed from nc30 to create an absolute file (AA.abs).

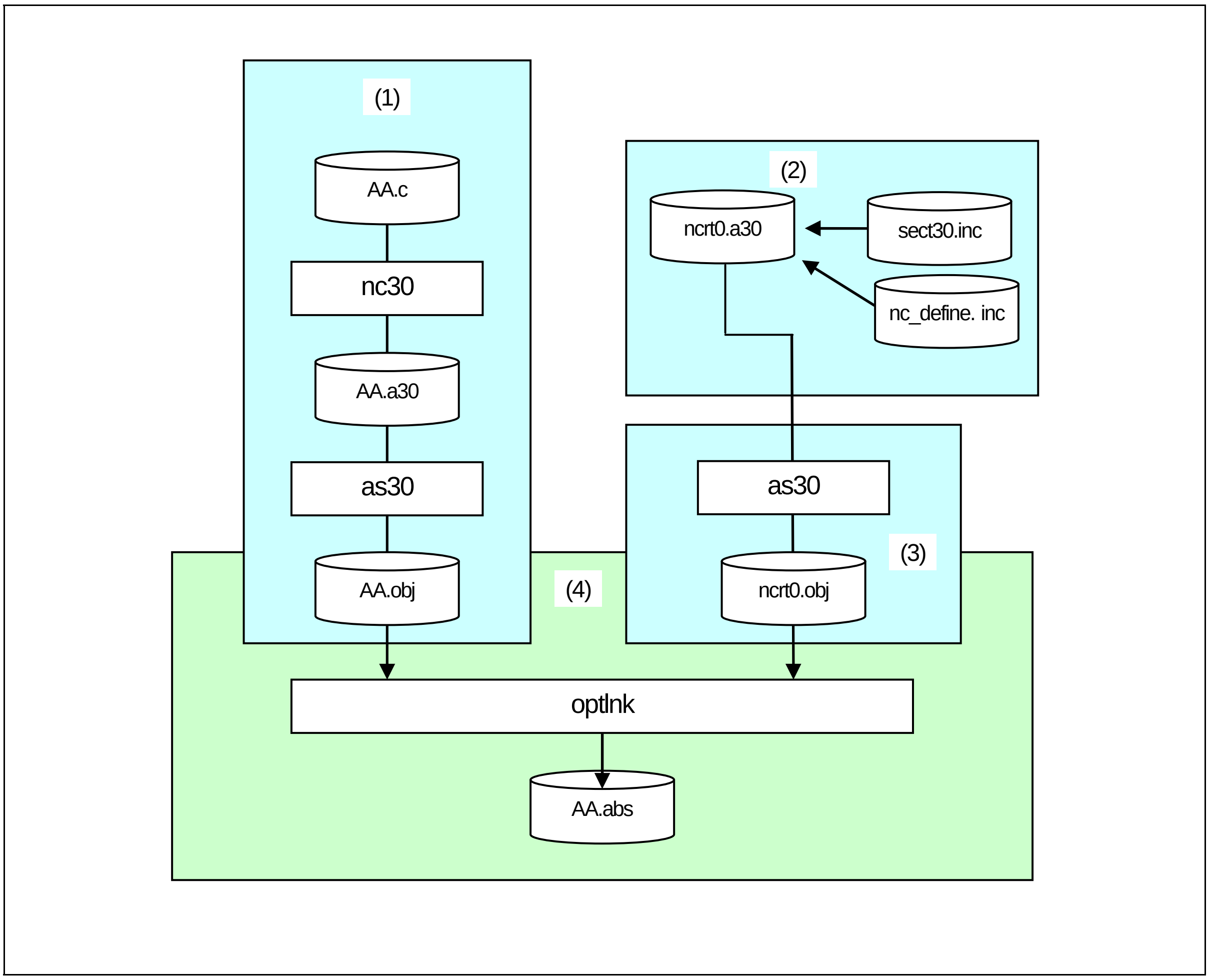


Figure1.3 Program Development Flow

Figure1.3 is an example make file containing the series of operations shown in Figure1.4.

```
AA.abs: ncrt0.obj AA.obj
            nc30 -oAA ncrt0.obj AA.obj

ncrt0.obj: ncrt0.a30
            as30 ncrt0.a30

AA.obj: AA.c
            nc30 -c AA.c
```

Figure1.4 Example make File

Figure1.5 shows the command line required for NC30 to perform the same operations as in the make file shown in Figure1.4.

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

%: Indicates the prompt
<RET>: Indicates the Return key
```

Figure1.5 Example NC30 Command Line

1.5 NC30 Output Files

The output files of NC30 are described here.

1.5.1 Introduction to Output Files

The compile driver nc30 outputs the files shown in Figure1.6 according to startup options. Beginning with the next page, the following describes the example files—and their displayed contents—that are output as a result of compile, assemble, and link processes performed on the C source file "sample.c" shown in Figure1.7. Note that if object files (.obj) created by compiling the source as a C++ program are moved to another directory before being linked, a problem may occur. Therefore, do not move the object files before linking.
For details about as30 and optlnk, see the Assembler and Optimizing Linkage Editor User’s Manual.

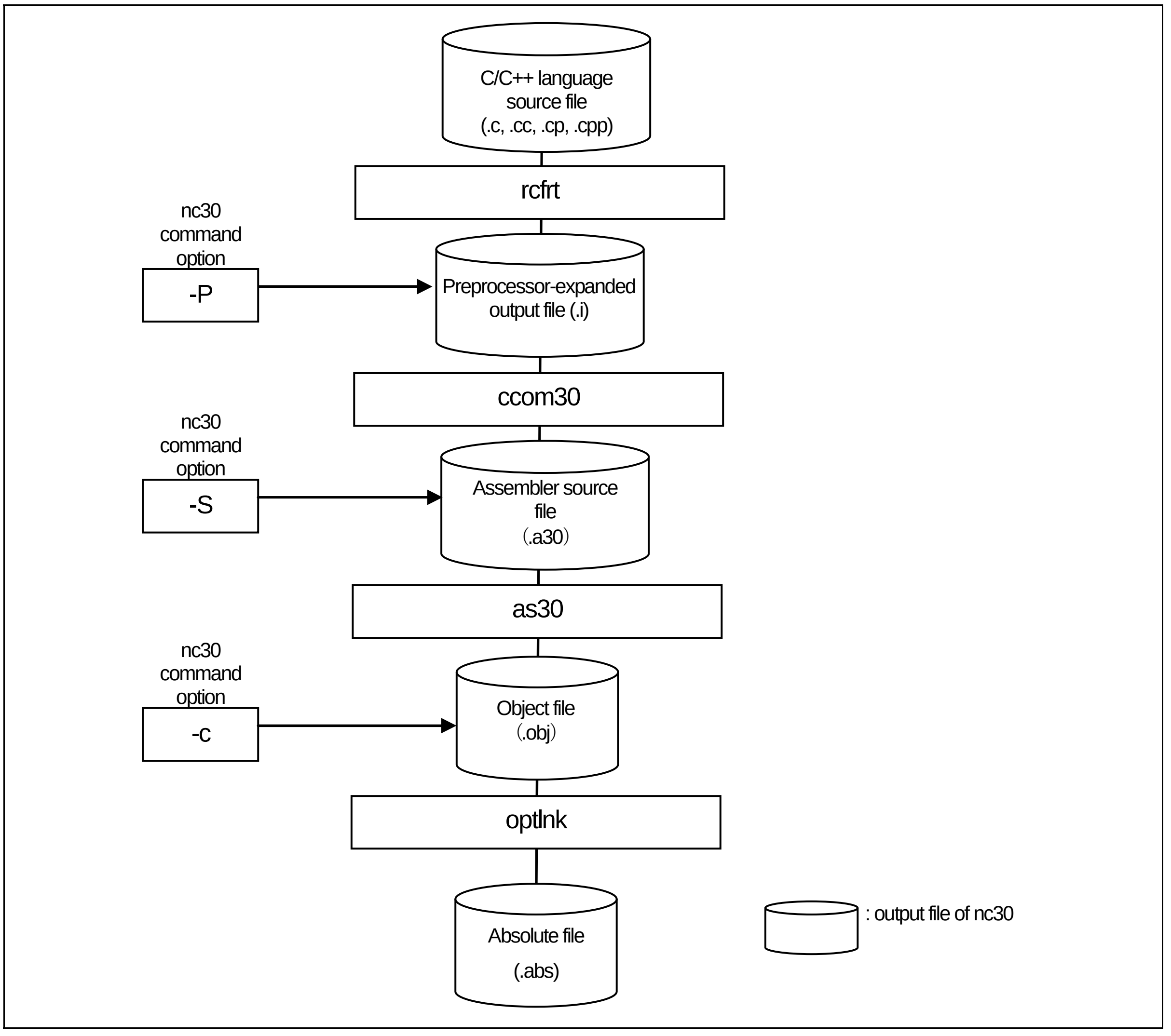


Figure1.6 Relationship of NC30 Command Line Options and Output Files

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

void        main(void)
{
            int         flag;

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

Figure1.7 Example C Source File (sample.c)

1.5.2 Preprocessor-Expanded Output Files

The preprocessor rcfrt performs such processing as to expand the contents of header files and macros, as well as make decision on conditional compilation, according to the preprocessing commands that begin with #.

The preprocessor-expanded output file contains the result of processing performed on C/C++ source files by rcfrt. Therefore, this file has only #pragma and #line but no other preprocessing lines output in it. By referring to the content of this file, it is possible to check the content of the program processed by the compiler. The file extension is ".i." Example output files are shown in Figure1.8 and Figure1.9.

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

typedef     unsigned int            size_t;

extern FILE _iob[];
```

Figure1.8 Example of a Preprocessor-Expanded Output File (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_t            fread(void _far *, size_t, size_t, FILE _far *);
                  :
               (omitted)
                  :
extern int     printf(const char _far *, ...);
extern int     fprintf(FILE _far *, const char _far *, ...);
extern int     sprintf(char _far *, const char _far *, ...);
                  :
               (omitted)
                  :
extern int     init_dev(FILE _far *, int);
extern int     speed(int, int, int, int);
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)

}
```

Figure1.9 Example of a Preprocessor-Expanded Output File (2)

The contents of preprocessor-expanded output files are described below. Paragraphs (1) through (4) below respectively correspond to the numbers (1) through (4) in Figure1.8 and Figure1.9.

(1) Shows the expansion of header file stdio.h specified in #include.

(2) Shows the C source program resulting from expanding the macro.

(3) Shows that CLR specified in #define is expanded as 0.

(4) Shows that since a PRN specified by #define is defined, the compile condition is met and, therefore, a printf function is output.

1.5.3 Assembly Language Source Files

This file is the one that has been converted from a preprocessor-expanded output file into an AS30-processible assembler source by the compiler body ccom30. The files output here are the assembler source file identified by the extension ".a30." Example output files are shown in Figure1.10 and Figure1.11.

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

;##     C Compiler                  OUTPUT
;## ccom30 Version X.XX.XX.XXX
;## Copyright (C) XXXX (XXXX - XXXX) Renesas Electronics Corporation.
;## and Renesas Solutions Corp. All rights reserved.
;## Compile Start Time XXX XXX XX XX:XX:XX XXXX

;## COMMAND_LINE: ccom30   -finfo -gnone -dS -o sample.a30 sample.i

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

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

            .SECTION  program,CODE,align
            ._inspect 'U', 2, "program", "program", 0
            ._file    'sample.c'
            ._inspect 'F', 's', "main", "_main", 'G', 7
            ._block   1h,1h
            ._line    6
;## # C_SRC :          {
            .glb      _main
_main:
            enter     #02H
            ._line    9
;## # C_SRC :                    flag = CLR;
            mov.w     #0000H,-2[FB]          ;   flag
            ._line    11
```

Figure1.10 Example of an Assembler Source File (1/2) (sample.a30)

```
;## # C_SRC :                 printf( "flag = %d¥n", flag );             ← (2)
          push.w    -2[FB]     ;   flag
          ._inspect 'S', 'p', 2
          push.w    #___T0>>16
          push.w    #(___T0&0FFFFH)
          ._inspect 'S', 'p', 4
          ._inspect 'S', 'c', "printf", "_printf", 'G', 0, 11
          ._block   2h,2h
          jsr       _printf
          ._eblock  2h,3h
          ._inspect 'S', 'p', -6
          add.b     #06H,SP
          ._line    13
;## # C_SRC :       }
          exitd
E1:
          .align
          ._eblock  1h,4h
          :
          (omitted)
          :
          .glb      _printf
          :
          (omitted)
          :
          .SECTION  rom_FO,ROMDATA
          ._inspect 'U', 4, "rom_FO", "rom_FO", 0
___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
          :
          (omitted)
          :
          .END

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

Figure1.11 Example of an Assembler Source File (2/2) (sample.a30)

The contents of assembler source files are described below. Paragraphs (1) through (2) below correspond to the numbers (1) through (2) in Figure1.10.

(1) Shows status of optimization option, and information on the initial settings of the near and far attribute for ROM and RAM.

(2) The contents of the source file are shown by comments.

1.5.4 Temporary Files Used by the Compiler

The compiler internally uses temporary files. Therefore, be aware that the following files, if present, are overwritten or removed.

- Source file name + extension ".fnm"
- Source file name + extension ".db1"
- Source file name + extension ".db2"
- Source file name + extension ".dbs"

Chapter 2 Basic Method for Using the Compiler

This chapter describes how to start the compile driver and the functionality of its startup options. The explanation of startup options here also includes the startup options of the assembler and the optimizing linkage editor that can both be started from the compile driver.

2.1 Starting Up the Compiler

2.1.1 Command Input Format of the Compile Driver

The compile driver activates each module of the compiler and the assembler and optimizing linkage editor. To activate this compile driver, the following information (input parameters) are required.

(1) C/C++ source file
(2) Assembler source file
(3) Object file
(4) Startup options (optional item)

Enter these items on the command line. At least one of the items (1), (2) or (3) need to be entered.

Figure 2.1 shows the input format. Figure 2.2 shows an example of how to enter.
In this example, the following is performed.

(1) Startup program ncrt0.a30 is assembled.
(2) C/C++ source program sample.c is compiled and then assembled.
(3) Object files ncrt0.obj and sample.obj are linked.

And an example of how to write the command line to create the absolute file sample.abs is shown.
The following startup options are used.

- Specifies the absolute file name sample.abs　　-o option
- Specifies output of list files (extension .lst) at assembling　　-as30 option
- Specifies output of list files (extension .map) at linking　　-lnkcmd option

```
%nc30△[startup option]△<[assembler source file name]△
                        [object file name]△[C/C++ source file name]>

%:    Prompt
< >:  Mandatory item
[ ]:  Optional item
△:    Space
```

Figure 2.1　Compile Driver's Command Input Format

```
% nc30 -osample -as30 "-l" -lnkcmd=command.txt ncrt0.a30 sample.c <RET>

<RET>: Return key
```

Figure 2.2 Compile Driver's Command Input Example

The files (assembler source files, object files, and C/C++ source files) whose name specifications except the path are the same cannot simultaneously be specified as input files for the compiler.

2.1.2 Command File

The command driver can compile a file which has multiple command options written in it (i.e., command file) after loading it.
Use of a command file makes it possible to circumvent Microsoft Windows limitations on the number of characters per command line.

a. Command file input format

```
%nc30△[startup option]△<@file name>△[startup option]

%:    Prompt
< >:  Mandatory
[ ]:  Optional
△:    Space
```

Figure 2.3 Command File Input Format

```
% nc30    @test.cmd    <RET>

<RET>: Return key
```

Figure 2.4 Command File Input Example

Command files are written in the manner described below.

```
ncrt0.a30
sample1.c sample2.obj
-osample
-lnkcmd=command.txt
```

Figure 2.5 Example of Command File Description

b. Rules on command file description

The following rules apply for command file description.

- Only one command file can be specified at a time. You cannot specify multiple command files simultaneously.
- No command files can be specified in another command file.
- Multiple command lines can be written in a command file.
- New-line characters in a command file are replaced with space characters.
- The maximum number of characters that can be written in one line of a command file is 2,048. An error results when this limit is exceeded.

c. Precautions to be observed when using a command file

A directory path can be specified for command file names. An error results if the file does not exist in the specified directory path.

You cannot specify two or more command files simultaneously. If multiple files are specified, the compiler displays an error message "Too many command files" before quitting the session.

2.1.3 Notes on Startup Options

a. Notes on writing startup options

The compile driver startup options are discriminated according to whether they are written in uppercase or lowercase letters. If an option is entered in the wrong case, the compile driver outputs a warning and continues processing, with the option assumed to be unspecified.

b. Priority of options for controlling the compile driver

Options for controlling the compile driver are subject to the following priority.

```
-E          -P          -S          -c
 ← High          Priority          Low →
```

For example, if the options,

- "-c" that creates an object file (extension ".obj") and finishes processing
- "-S" that creates an assembler source file (extension ".a30") and finishes processing

are specified at the same time, then the "-S" option has priority. In this case, the compile driver terminates after the assembler finishes processing. Only assembler source files are generated.

To generate object files and also assembler source files at the same time, use the option "-dsource" (or "-dS" in short form).

c. Notes on invoking optlnk from the compile driver

The compile driver, when invoking optlnk, automatically adds the following options before activating it. Be aware that these options, if specified, cause a warning against duplicates and are ignored.

-nologo, -library=nc30lib.lib, -output=<file name>, -total_size

Also, the compile driver, when invoking optlnk, automatically, or after modification, adds the following options according to user-specified options before activating it. Be aware that these options, if specified by user, cause a warning against duplicates and are ignored.
The -fno_lib option helps to suppress specification for linking the standard library.
Note, however, that if optlnk needs to be started directly, the standard library, etc. must be linked properly.

User-specified options	Options passed to optlnk by the compile driver
-dir	-output=<directory name>\<file name>
-o	-output=<file name>
-OS_MAX	-optimize=branch
-OR_MAX	-optimize=branch
-g	-debug -list -show=all
-fsizet_16	-library="%LIB30%\nc30s16.lib"
-fptrdifft_16	-library="%LIB30%\nc30s16.lib"
-R8C	-library="%LIB30%\r8clib.lib"
-R8C -fsizet_16	-library="%LIB30%\r8cs16.lib"
-R8C -fptrdifft_16	-library="%LIB30%\r8cs16.lib"
-R8CE	-library="%LIB30%\r8celib.lib"
-R8CE -fsizet_16	-library="%LIB30%\r8ces16.lib"
-R8CE -fptrdifft_16	-library="%LIB30%\r8ces16.lib"
-finfo	-debug -list -show=all
-fSB_auto	-debug -list -show=all
-Wstop_at_link	-change_message=error
-Wno_used_function	-message -msg_unused

2.1.4 nc30 Startup Options

a. Options for controlling the compile driver

Tables 2.1 and 2.2 list the startup options that relate to control of the compile driver. For notes on each option and other details, see Appendix A.

Table 2.1 Compile Driver Control Options (1/2)

Option	Function
-c	Creates a object file (extension .obj) and finishes processing[1].
- D *identifier name*	Defines an identifier. Same as #define.
-dsource (shortcut -dS)	Generates an assembler source file (extension ".a30"). Do not assemble the assembler source files generated by this option.
-dsource_in_list (shortcut -dSL)	Generates an assembler list file (".lst").
-E	Processes only preprocess commands and outputs the result to standard output.
-I *directory name*	Specifies the directory in which to search for the files referenced by the preprocess command #include.
-P	Processes only preprocess commands and creates a file (extension ".i").
-S	Creates an assembler source file (extension .a30) and finishes processing. Note that template functions are output as static functions in the .a30 file. Be aware that if the assembler source files generated by this option are assembled, C/C++ level debug information is lost.
-silent	Suppresses output of copyright messages at startup.
-U *predefined macro name*	Undefines the specified predefined macro.
-lang={c|cpp|ecpp}	Specifying c lets the driver compile the input file as C (C89) source file. Specifying cpp lets the driver compile the input file as C++ source file. Specifying ecpp lets the driver compile the input file as EC++ source file. -exception and -rtti cannot be selected simultaneously with ecpp. If, when these options are omitted, the input file has the extension .cpp, .cc, or .cp, it is compiled as C++ source file. Otherwise, it is compiled as C (C89) source file. However, if the input file bears the extension .a30, it is handled as an assembler source file regardless of whether these options are specified.

[1] Unless the startup options -c, -E, -P, or -S are specified, nc30 only performs control till optlnk, finishing the process after creating absolute files (extension .abs).

Table 2.2 Compile Driver Control Options (2/2)

Option	Function
-preinclude=file name[・・・]	Includes the specified file. If there are multiple files, they can be specified by separating each with a comma. In that case, files are searched in order, from left to right. If multiple instances of this option are specified, all of the specified files are included. Note that files are searched in the same order as specified by #include "file name."
-exception	Enables the exception processing feature. If compiled as C (C89), this option incurs a warning and has no effect. It cannot be selected simultaneously with -lang=ecpp. If you specify this option, use the standard library that was generated by the library generator with this option added.
-noexception	Disables the exception processing feature. -noexception is the default.
-rtti=on	Enables the C++ runtime type information feature (dynamic_cast, typeid). If compiled as C (C89), this option incurs a warning and has no effect. Cannot be selected simultaneously with -lang=ecpp. If you specify this option, use the standard library that was generated by the library generator with this option added.
-rtti=off	Disables the C++ runtime type information feature (dynamic_cast, typeid). -rtti=off is the default. Cannot be selected simultaneously with -lang=ecpp.

b. Options for specifying output files

Table 2.3 lists the startup options that specify the names of absolute files.

Table 2.3 Options for Specifying Output Files

Option	Function
-dir *directory name*	Specifies the destination directory for the files generated by the compiler.
-o *file name*	Specifies the name of the file generated by optlnk. It is also possible to specify a path name that includes a directory name. Do not specify the filename extension.

c. Options for displaying version and command line information

Table 2.4 lists the startup options for displaying the versions of the cross tools used and the command line.

Table 2.4 Options for Displaying Version and Command Line Information

Option	Function
-v	Displays the command program names under execution and the command line.
-V	Only displays the startup message of each compiler program and finishes processing (without compiling).

d. Debug options

Table 2.5 lists the startup options for debugging to output C/C++ level debug information.

Table 2.5 Debug Options

Option	Function
-g	Outputs debug information to the object file.
-genter	Always outputs an enter instruction when calling a function. Be sure to specify this option when using the debugger's stack trace feature.
-gno_reg	Suppresses output of debug information on register variables.

e. Optimization options

Table 2.6 lists the startup options for optimization to maximize the program's execution speed and minimize the ROM capacity.

Table 2.6 Optimization Options

Option	Short form	Function
-O[1～5]	None	Performs optimization that is effective for both speed and ROM size at each level.
-OR	None	Performs ROM size-oriented optimization.
-OS	None	Performs speed-oriented optimization.
-OR_MAX	-ORM	Performs ROM size-prioritized, maximum optimization.
-OS_MAX	-OSM	Performs the highest speed-oriented optimization possible.
-Ocompare_byte_to_word	-OCBTW	Compares bytes at contiguous addresses wordwise.
-Oconst	-OC	Optimizes compilation by replacing references to the const-qualified external variables with constants.
-Oforward_function_to_inline	-OFFTI	Expands all inline functions in-line.
-Oloop_unroll[=loop count]	-OLU	Unrolls code as many times as the loop count without revolving the loop statement. The "loop count" can be omitted. When omitted, this option is applied to a loop count of up to 5.
-Ono_asmopt	-ONA	Suppresses optimization by the assembler optimizer "aopt30."
-Ono_bit	-ONB	Suppresses optimization based on grouping of bit manipulations.
-Ono_break_source_debug	-ONBSD	Suppresses optimization that affects source line information.
-Ono_float_const_fold	-ONFCF	Suppresses the constant folding processing of floating-point numbers. This optimization is effective only when compiled as C program.
-Ono_logical_or_combine	-ONLOC	Suppresses optimization that puts logical Ors together.
-Ono_stdlib	-ONS	Inhibits inline padding of standard library functions and modification of library functions.
-Osp_adjust	-OSA	Optimizes removal of stack correction code. This makes it possible to reduce the ROM capacity. However, it may result in an increased amount of stack used.
-Ostack_frame_align	-OSFA	Aligns the stack frame on an even address boundary.
-Ostatic_to_inline	-OSTI	Handles static-declared functions as inline declared.
-O5OA	None	Suppresses generation of codes using bit-manipulating instructions (BTSTC, BTSTS) when the optimization option "-O5" is selected.
-goptimize	None	Outputs additional information for inter-module optimization.

f. Options for modifying generated code

Table 2.7 lists the startup options to control the assembly language generated by this compiler.

Table 2.7 Generated Code Modification Options (1/3)

Option	Short form	Function
-fansi	None	Makes reserved words and arithmetic operation methods ANSI-compliant. When compiled as C++ program, asm and inline are made the keywords and char-type data is promoted to type int when the data is operated on, regardless of whether this option is specified.
-fchar_enumerator	-fCE	Handles the type of enumerator as type unsigned char, and not as type int.
-fconst_not_ROM	-fCNR	Does not handle the type specified by const as ROM data.
-fdouble_32	-fD32	Processes type double as type float. If this option is used, overloaded definitions of types float and double in a C++ program are prohibited. When this option is added, -Wnon_prototype is enabled at the same time.
-fenable_register	-fER	Enables register storage class.
-fextend_to_int	-fETI	Promotes char data to type int when the data is operated on (promoted as stipulated in ANSI standard)[2]. This option is usable for C (C89). When used for C++, this option incurs a warning and is ignored. When compiled as C++ program, char data is always promoted to type int when the data is operated on.
-ffar_RAM	-fFRAM	Changes the default attribute of RAM data to far.
-finfo	None	Outputs inspector information.
-fbit	-fB	Generates code assuming that bitwise manipulating instructions can be used for all external variables mapped to the near area.
-fno_carry	-fNC	Suppresses carry flag addition when data is indirectly accessed using far pointers.
-fauto_128	-fA1	Limits the maximum size of stack frames used to 128 bytes.
-ffar_pointer	-fFP	Changes the default attribute of pointer type variables to far. The pointer variables defined by the near qualifier always assume the near attribute regardless of whether this option is specified.
-fnear_ROM	-fNROM	Changes the default attribute of ROM data to near.
-fno_align	-fNA	Does not align the start address of the function concerned.
-fno_even	-fNE	When outputting data, locates all of them in sections that have the odd attribute without separating between odd data and even data.
-fno_switch_table	-fNST	Performs comparison on switch statement before generating branch code.
-fnot_address_volatile	-fNAV	Does not handle the variables specified with #pragma ADDRESS as the ones specified with volatile.
-fnot_reserve_asm	-fNRA	Excludes asm from reserved words. (Only "_asm" is valid.) When compiled as C++ program, asm is made a keyword regardless of whether this option is specified.

[2] Under ANSI standard, char data or signed char data is always promoted to type int when the data is evaluated.
This is because operations on char types (e.g., c1=c2*2/c3;) would otherwise cause the char type to overflow in the middle of operation, producing an unexpected result.

Table 2.8 Generated Code Modification Options (2/3)

Option	Short form	Function
-fnot_reserve_far_and_near	-fNRFAN	Excludes far and near from reserved words. (Only far and _near are valid.)
-fnot_reserve_inline	-fNRI	Excludes inline from reserved words. (Only _inline is made a reserved word.) When compiled as C++ program, inline is made a keyword regardless of whether this option is specified.
-fsmall_array	-fSA	When referring to a far-type array whose total size is unknown, the compiler assumes that its total size is within 64K bytes and calculates the subscripts in 16 bits.
-fswitch_other_section	-fSOS	Outputs a jump table for switch statements to a different section than the program section.
-fchange_bank_always	-fCBA	Switches the bank every time.
-fauto_over_255	-fAO2	Changes the maximum stack frame size that can be reserved in one function to 64 Kbytes.
-fsizet_16	-fS16	Changes the type definition size_t from unsigned long type to unsigned int type. If you specify this option, select the standard library nc30s16.lib, r8cs16.lib (when-R8C specified), or r8ces16.lib (when -R8CE specified). Otherwise, use the standard library that was generated by the library generator with this option added.
-fptrdifft_16	-fP16	Changes the type definition ptrdiff_t from signed long type to signed int type. If you specify this option, select the standard library nc30s16.lib, r8cs16.lib (when-R8C specified), or r8ces16.lib (when -R8CE specified). Otherwise, use the standard library that was generated by the library generator with this option added.
-fuse_DIV	-fUD	Changes generated code for division.
-fuse_MUL	-fUM	Changes generated code for multiplication.
-R8C	None	Generates code appropriate for the R8C family MCU. When this option is specified, the keywords far and _far are ignored. If you specify this option, select the standard library r8clib.lib or r8cs16.lib (when -fsizet_16 or -fptrdifft_16 specified). Otherwise, use the standard library that was generated by the library generator with this option added. Add this option for all programs to be linked.
-R8CE	None	Generates code appropriate for the R8C family MCU (64K ROM or larger). If you specify this option, select the standard library r8celib.lib or r8ces16.lib (when -fsizet_16 or -fptrdifft_16 specified). Otherwise, use the standard library that was generated by the library generator with this option added. Add this option for all programs to be linked.

Table 2.9 Generated Code Modification Options (3/3)

Option	Short form	Function
-fSB_auto	-fSBA	Automatically generates SB relative addressing for all functions. The number of times external variables are referenced in each function are analyzed to generate optimum SB relative addressing. (1) The address of a symbol that serves as the base point for SB relative is stored in the SB register. (2) Code for saving and restoring the SB register at entry to and exit from the function is generated. (3) This option applies for only external variables. (4) This option cannot be used in combination with -OR, -OS, -OR_MAX(-ORM), and -OS_MAX(-OSM). (5) If object files using this option and those using the following features are linked to build a program, behavior of the program cannot be guaranteed. ● #pragma SBDATA ● Compiler option-fauto_over_255(-fAO2)

g. Library specifying options

Table 2.10 lists the startup options for specifying a library file.

Table 2.10 Library Specifying Options

Option	Function
-l library file name	Specifies the library to be used when linking.
-fno_lib	Suppresses the facility to automatically add the -library option when invoking optlnk from the compile driver.

h. Warning options

Table 2.11 lists the startup options for outputting warning messages for contraventions of language specifications of this compiler.

Table 2.11 Warning Options

Option	Short form	Function
-Wall	None	Shows all detectable warnings (except those that are output for "-Wlarge_to_small" and "-Wno-used_argument").
-Wccom_max_warnings =*warning count*	-WCMW	Allows you to specify an upper limit for the number of warnings output by ccom30. This facility works only when compiled as C program.
-Wlarge_to_small	-WLTS	Outputs a warning for implicit assignments from large size to small size.
-Wnesting_comment	-WNC	Outputs a warning when a "/*" is written in a comment.
-Wno_stop	-WNS	Prevents compile operation from stopping when an error occurs. When compiled as C++ program, operation stops when 100 errors have been output, regardless of whether this option is selected.
-Wno_used_argument	-WNUA	Outputs a warning for unused arguments in a defined function that has arguments.
-Wno_used_function	-WNUF	Displays unused global functions when linking. This option is unnecessary when the linker options -msg_unused and -message are used.
-Wno_used_static_function	-WNUSF	Displays the static function names that do not require code generation.
-Wno_warning_stdlib	-WNWS	If you specify this option while "-Wnon_prototype" or "-Wall" is specified, the compiler suppresses "warnings for a standard library where function prototypes are not declared." When compiled as a C++ program, regardless of whether this option is specified, the compiler outputs a message whenever function prototype declarations are nonexistent.
-Wnon_prototype	-WNP	Outputs a warning if functions without prototype declarations are used. When compiled as a C++ program, regardless of whether this option is specified, the compiler outputs a message whenever function prototype declarations are nonexistent.
-Wstop_at_link	-WSAL	Suppresses generation of absolute files if a warning occurs at link time.
-Wstop_at_warning	-WSAW	Stops compile process when a warning occurs.
-Wundefined_macro	-WUM	Warns when an undefined macro is used in #if.
-Wuninitialize_variable	-WUV	Outputs a warning for auto variables that have not been initialized.
-Wunknown_pragma	-WUP	Outputs a warning when unsupported #pragma is used.

i. Assemble and link options

Table 2.12 lists the startup options for specifying as30 and optlnk options.

Table 2.12 Assemble and Link Options

Option	Function
-as30△<option>	Specifies options for the assemble command as30. If two or more options need to be passed, enclose them in double quotes.
-lnkcmd=<file name>	Specifies a command file for optlnk.

2.2 Preparing the Assembler Startup Program

For programs written in C/C++ to be 'burned' into ROM, a startup program written in assembly language or C to initialize the microcontroller, locate sections, and set up interrupt vector tables, etc. is required. The startup program needs to be modified to suit the MCU type you're using and the system in which it is used. This section describes an assembler startup program written in assembly language and how to customize it. Note that when, after launching the integrated development environment (High-performance Embedded Workshop), you select Application for the project type in creating a new project, a template for assembler startup programs is automatically generated in a folder . Modify this template to suit your need.

2.2.1 Sample of the Assembler Startup Program

The assembler startup program consists of the following three files:

- ncrt0.a30
  Write a program that is executed immediately after reset.
- nc_define.inc
  This file defines the sizes of the stack and heap areas and the addresses of the variable vectorand special-page vector.
- sect30.inc
  Included from ncrt0.a30, this file defines section locations (memory mapping).

The source program list of ncrt0.a30 is shown below.

```
;/********************************************************************
;*
;* Device       : M16C/60,30,20,10
;*
;* File Name    : ncrt0.a30
;*
;* Abstract     : Startup Program for M16C/60,30,20,10.
;*
;* History      : X.XX    (xxxx-xx-xx)
;*
;* Copyright(c) 2010 Renesas Electronics Corporation
;*                   And Renesas Solutions Corp.,All Rights Reserved.
;*
;********************************************************************/

;---------------------------------------------------------------------
; include files
;---------------------------------------------------------------------
                .list           OFF
                .include        nc_define.inc
                .include        sect30.inc                      ← (1)
                .list           ON

;---------------------------------------------------------------------
; BankSelect definition for 4M mode
;---------------------------------------------------------------------
;               .glb            __BankSelect
;__BankSelect   .equ            0BH
```

Figure 2.6 Excerpt of the Assembler Startup Program, ncrt0.a30 (1/5)

```
;=====================================================================
; Interrupt section start
;---------------------------------------------------------------------
                .glb            start
                .section interrupt,CODE,ALIGN
                .insf           start,G,0
start:                                                  ← (2)
;---------------------------------------------------------------------
; after reset,this program will start
;---------------------------------------------------------------------
                ldc             #((topof istack)+(sizeof istack)),isp ;set istack pointer
                mov.b           #02h,0ah
                mov.b           #00h,04h                ← (3)
                mov.b           #00h,0ah
.if __STACKSIZE__ != 0
                ldc             #0080h,flg              ← (4)
                ldc             #((topof stack)+(sizeof stack)),sp ;set stack pointer
.else
                ldc             #0000h,flg
.endif
                ldc             #__SB__,sb              ;set sb register

                ; If the destination is INTBL or INTBH,
                ; make sure that bytes are transferred in succession.
                ldc             #((topof vector)>>16)&0FFFFh,INTBH  ← (5)
                ldc             #(topof vector)&0FFFFh,INTBL

;=====================================================================
; NEAR area initialize.
;---------------------------------------------------------------------
; bss zero clear                                        ← (6)
;---------------------------------------------------------------------
                N_BZERO         (topof bss_SE),bss_SE
                N_BZERO         (topof bss_SO),bss_SO
                N_BZERO         (topof bss_NE),bss_NE
                N_BZERO         (topof bss_NO),bss_NO
```

Figure 2.7 Excerpt of the Assembler Startup Program, ncrt0.a30 (2/5)

```
;---------------------------------------------------------------
; initialize data section                                        ← (7)
;---------------------------------------------------------------
                N_BCOPY         (topof data_SEI),(topof data_SE),data_SE
                N_BCOPY         (topof data_SOI),(topof data_SO),data_SO
                N_BCOPY         (topof data_NEI),(topof data_NE),data_NE
                N_BCOPY         (topof data_NOI),(topof data_NO),data_NO

;====================================================================
; FAR area initialize.
;---------------------------------------------------------------
; bss zero clear                                                 ← (8)
;---------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                BZERO           (topof bss_FE),bss_FE
                BZERO           (topof bss_FO),bss_FO.
.endif

;---------------------------------------------------------------
; initialize data section                                        ← (9)
;---------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                BCOPY           (topof data_FEI),(topof data_FE),data_FE
                BCOPY           (topof data_FOI),(topof data_FO),data_FO
.if __STACKSIZE__ != 0
                ldc             #((topof stack)+(sizeof stack)),sp.
.else
                ldc             #((topof istack)+(sizeof istack)),isp.
.endif
                .stk            -40.
.endif

;====================================================================
; heap area initialize                                           ← (10)
;---------------------------------------------------------------
.if __HEAPSIZE__ != 0
                .glb            __mnext
                .glb            __msize
                mov.w           #((topof heap_NE)&0FFFFH),__mnext
                mov.w           #((topof heap_NE)>>16),__mnext+2
                mov.w           #(__HEAPSIZE__&0FFFFH),__msize
                mov.w           #(__HEAPSIZE__>>16),__msize+2.
.endif
```

Figure 2.8 Excerpt of the Assembler Startup Program, ncrt0.a30 (3/5)

```
;====================================================================
; Initialize standard I/O                                            ← (11)
;---------------------------------------------------------------------
.if __STANDARD_IO__ == 1
                .glb            __init
                .call           __init,G
                jsr.a           __init
.endif

;====================================================================
; Call main() function                                               ← (12)
;---------------------------------------------------------------------
                ldc             #0h,fb                  ; for debuger

; Remove the comment when you use global class object               ← (13)
; Sections C$INIT will be generated
;               .glb            __CALL_INIT
;               .call           __CALL_INIT,G
;               jsr.a           __CALL_INIT

                .glb            _main
                .call           _main,G
                jsr.a           _main

;====================================================================
; exit() function                                                    ← (14)
;---------------------------------------------------------------------
                .glb            _exit
                .glb            $exit
                .glb            __exit_loop
_exit:
$exit:

; Remove the comment when you use global class object               ← (15)
; Sections C$INIT will be generated
;               .glb            __CALL_END
;               .call           __CALL_END,G
;               jsr.a           __CALL_END

__exit_loop:                                            ; End program
                jmp             __exit_loop
                .einsf
```

Figure 2.9 Excerpt of the Assembler Startup Program, ncrt0.a30 (4/5)

```
;======================================================================
; dummy interrupt function
;----------------------------------------------------------------------
                    .glb            dummy_int                    ← (16)
dummy_int:
                    reit

                    .end
```

(1) Includes sect30.inc.
(2) Starts from the label start immediately after reset.
(3) Sets processor operation mode.
(4) Sets the interrupt priority level and various flags.
(5) Defines the start address of the interrupt vector table.
(6) Clears the bss section in the near area to zeros.
(7) Transfers the initial value of the data section in the near area to the RAM area.
(8) Clears the bss section in the far area to zeros
(9) Transfers the initial value of the data section in the far area to the RAM area.
(10) Initializes the heap area. Comment out this line when no memory management functions are used.
(11) Calls the init function that initializes standard I/O. Comment out this line when no I/O functions are used.
(12) Use of global class objects (C++) may result in a C$INIT section being generated. In that case, remove this comment and then relink.
(13) Calls the main function.
  * Interrupts are disabled when the main function is called. To use the interrupts, enable them with the FSET instruction.
(14) This is an exit function part.
(15) Use of global class objects (C++) may result in a C$INIT section being generated. In that case, remove this comment and then relink.
(16) This is a dummy interrupt processing function.

Figure 2.10 Excerpt of the Assembler Startup Program, ncrt0.a30 (5/5)

Next, the source program list of nc_define.inc is shown below.

```
__FAR_RAM_FLG__     .equ    0       ; FAR RAM flag definition
__STANDARD_IO__     .equ    0       ; STANDARD I/O flag definition
__HEAPSIZE__        .equ    0300H   ; HEEP SIZE definition
__STACKSIZE__       .equ    0300H   ; STACK SIZE definition
__ISTACKSIZE__      .equ    0300H   ; INTERRUPT STACK SIZE definition
```

Figure 2.11 Excerpt of the Assembler Startup Program, nc_define.inc

Next, the source program list of sect30.inc is shown below.

```
;/*********************************************************************
;*
;* Device       : M16C/60,30,20,10
;*
;* File Name    : sect30.inc
;*
;* Abstract     : Section definition for M16C/60,30,20,10
;*
;* History      : x.xx   (xxxx-xx-xx)
;*
;* Copyright(c) 2010 Renesas Electronics Corporation
;*                  And Renesas Solutions Corp.,All Rights Reserved.
;*
;*********************************************************************/

;====================================================================
;
;               Definition of section
;
;--------------------------------------------------------------------
; Near RAM data area
;--------------------------------------------------------------------
; SBDATA area
                .section    data_SE,DATA,ALIGN
                .section    bss_SE,DATA,ALIGN
                .section    data_SO,DATA
                .section    bss_SO,DATA

; SBDATA area definition                                    ← (1)
; Sets the top address (__SB__) of the SBDATA area
; (it is accessing area to used the SBrelative addressing mode).
                .glb                    __SB__
__SB__      .equ            400H

; near RAM area
                .section    data_NE,DATA,ALIGN
                .section    bss_NE,DATA,ALIGN
                .section    data_NO,DATA
                .section    bss_NO,DATA

;--------------------------------------------------------------------
; Stack area
;--------------------------------------------------------------------
.if __STACKSIZE__ != 0
                .section    stack,DATA,ALIGN
                .blkb                   __STACKSIZE__       ← (2)
.endif
```

Figure 2.12 Excerpt of the Assembler Startup Program List, sect30.inc (1/5)

```
                .section    istack,DATA,ALIGN
                .blkb               __ISTACKSIZE__              ← (3)

;---------------------------------------------------------------
; heap section
;---------------------------------------------------------------
.if __HEAPSIZE__ != 0
                .section    heap_NE,DATA,ALIGN
                .blkb               __HEAPSIZE__                ← (4)
.endif

;---------------------------------------------------------------
; Far RAM data area
;---------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                .section    data_FE,DATA,ALIGN
                .section    bss_FE,DATA,ALIGN
                .section    data_FO,DATA
                .section    bss_FO,DATA
.endif

;---------------------------------------------------------------
; Initial data of 'data' section
;---------------------------------------------------------------
                .section    data_SEI,ROMDATA
                .section    data_SOI,ROMDATA
                .section    data_NEI,ROMDATA
                .section    data_NOI,ROMDATA
.if __FAR_RAM_FLG__ != 0
                .section    data_FEI,ROMDATA
                .section    data_FOI,ROMDATA
.endif

;---------------------------------------------------------------
; variable vector section
;---------------------------------------------------------------
                .section    vector,ROMDATA

; When you use "#pragma interrupt" with "vect=",                ← (5)
; you need not define interrupt vector.
;
; When you use "#pragma interrupt" without "vect=",
; you must define all interrupt vectors like the following example.
; You define dummy_int for interrupt vector not used.
```

Figure 2.13 Excerpt of the Assembler Startup Program List, sect30.inc (2/5)

```
;                       .lword      dummy_int           ; vector    0
;                       .lword      dummy_int           ; vector    1
;                       .lword      dummy_int           ; vector    2
;                                        :
;                       .lword      dummy_int           ; vector 63

;---------------------------------------------------------------
; for User Boot Code Area
; Please custumize this data for your setting.
;---------------------------------------------------------------
.if 0
            .section _UB_section_FE,ROMDATA
            .org    013ff0H
            .byte 0FFh,0FFh,0FFh,0FFh,0FFh,0FFh,0FFh,0FFh ; User boot code
            .word 0FFFFh                                   ; Port address
            .byte 0FFh                                     ; Port bit
            .byte 0FFh                                     ; Boot level
            .byte 0FFh,0FFh,0FFh,0FFh                      ; Reserved
.endif

;---------------------------------------------------------------
; fixed vector section
;---------------------------------------------------------------
                        .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
```

Figure 2.14 Excerpt of the Assembler Startup Program List, sect30.inc (3/5)

```
;==================================================================
; ID code & ROM code protect
;------------------------------------------------------------------
; ID code check function
                    .id "#FFFFFFFFFFFFFF"

; ROM code protect control address
                    ; .protect 00H

;==================================================================
; Initialize Macro declaration
;------------------------------------------------------------------
N_BZERO .macro          TOP_,SECT_
                    mov.b           #00H,R0L
                    mov.w           #(TOP_ & 0FFFFH),A1
                    mov.w           #sizeof SECT_,R3
                    sstr.b
                    .endm

N_BCOPY .macro          FROM_,TO_,SECT_
                    mov.w           #(FROM_ & 0FFFFH),A0
                    mov.b           #(FROM_ >> 16),R1H
                    mov.w           #TO_,A1
                    mov.w           #sizeof SECT_,R3
                    smovf.b
                    .endm

BZERO   .macro          TOP_,SECT_
                    push.w          #sizeof SECT_ >> 16
                    push.w          #sizeof SECT_  & 0ffffh
                    pusha           TOP_ >> 16
                    pusha           TOP_ & 0ffffh
                    .stk            8
                    .glb            _bzero
                    .call           _bzero,G
                    jsr.a           _bzero
                    .endm

BCOPY   .macro          FROM_ ,TO_ ,SECT_
                    push.w          #sizeof SECT_  >> 16
                    push.w          #sizeof SECT_  & 0ffffh
                    pusha           TO_ >> 16
                    pusha           TO_  & 0ffffh
                    pusha           FROM_ >> 16
                    pusha           FROM_  & 0ffffh
                    .stk            12
                    .glb            _bcopy
                    .call   _bcopy,G
                    jsr.a           _bcopy
                    .endm
```

Figure 2.15 Excerpt of the Assembler Startup Program List, sect30.inc (4/5)

(1) Defines the start address of the SBDATA area.
(2) Defines the user stack size.
(3) Defines the interrupt stack size.
(4) Defines the heap area to be used.
(5) Defines the vector table.

Figure 2.16 Excerpt of the Assembler Startup Program List, sect30.inc (5/5)

2.2.2 Customizing the Assembler Startup Program

a. Overview of assembler startup program processing

(1) About ncrt0.a30

This program is run at the time of program start or immediately after reset.
It performs mainly the following process:

- Sets the processor operation mode.
- Initializes the stack pointers (ISP register and USP register).
- Initializes the SB register.
- Initializes the INTB register.
- Initializes the near area of data.
  The bss_SE, bss_SO, bss_NE, and bss_NO sections are cleared to 0.
  Also, the initial values of these sections stored in the ROM area (data_SEI, data_SOI, data_NEI, and data_NOI) are transferred to the RAM area (data_SE, data_SO, data_NE, and data_NO).
- Initializes the far area of data.
  The bss_FE and bss_FO sections are cleared to 0.
  Also, the initial values of these sections stored in the ROM area (data_FEI and data_FOI) are transferred to the RAM area (data_FE and data_FO).
- Initializes the heap area.
- Initializes the standard I/O function library.
- Makes a call for the dynamic initialization of static objects.
- Calls the main function.

b. Procedure for modifying the assembler startup program

The following shows the procedure for modifying the assembler startup program to make it appropriate for the system in which it will be incorporated.

- c. Examples of assembler startup modifications that require caution
- d. Setting the size of the stack section
- e. Setting the size of the heap area.
- f. Setting the interrupt vector table
- g. Setting the processor mode register

c. Examples of assembler startup modifications that require caution

(1) Settings when not using the standard I/O functions

The _init function[3] initializes input/output of the standard I/O function library. It is called before the main function is called in ncrt0.a30. Figure 2.17 shows a program part where the _init function is called.
If your application program does not use standard I/O functions, set the __STANDARD_IO__ macro within nc_define.inc to 0.

```
;======================================================================
; Initialize standard I/O
;---------------------------------------------------------------------
.if __STANDARD_IO__ == 1
            .glb        __init
            .call       __init,G
            jsr.a       __init
.endif
```

Figure 2.17 Part of ncrt0.a30 Where _init Function is Called

To use only the sprintf, vsprintf, and sscanf functions, there is no need to call the _init function. In this case, since the __sget, __iob, $_fp, or $_sput symbols may result in an undefined error at link time, create a dummy stub function before linking, as shown below.

~~.if __PROGRAM_NO_ALIGN__ ==1~~
~~.section program,code~~
~~.else~~
~~.section program,code,align~~
~~.endif~~
~~.glb $_fp, $_pc, __fs~~
~~$_fp:~~
~~$_pc:~~
~~__fs:~~
~~rts~~
~~.end~~

```
.if    __PROGRAM_NO_ALIGN__==1
   .section program,code
.else
   .section program,code,align
.endif
   .glb $__fp, $__pc, ___fs, ___sc
$__fp:
$__pc:
___fs:
___sc:
   rts
   .end
```

[3] The _init function also initializes the microcontroller (hardware) for standard I/O functions. To use the standard I/O functions, the _init function, etc. need to be corrected depending on the system in which the program is incorporated. The source file of the _init function is generated in a project that was created under the integrated development environment (High-performance Embedded Workshop) after selecting the project type "C Source Startup Application" and enabling the checkbox "Use the I/O Library."

(2) Settings when not using the memory management functions

In order to use the memory management functions (e.g., calloc and malloc), following settings are made in ncrt0.a30, in addition to reserving storage for the heap area.

(1) Initialization of the external variable char * __mnext
Initializes the start address of the heap area with the label topofheap_NE.

(2) Initialization of the external variable unsigned long__msize
Initializes with __HEAPSIZE__ that was set in Section 2.2.2, Paragraph e, "Sets the size of the heap area."

Figure 2.18 shows the initialization part in ncrt0.a30.

```
.if __HEAPSIZE__ != 0
  .glb   __mnext
  .glb   __msize
  mov.w  #((topof heap_NE)&0FFFFH),__mnext
  mov.w  #((topof heap_NE)>>16),__mnext+2
  mov.w  #(__HEAPSIZE__&0FFFFH),__msize
  mov.w  #(__HEAPSIZE__>>16),__msize+2
.endif
```

Figure 2.18 Initialization Part in ncrt0.a30 when Using Memory Management Functions

If your application program does not use memory-management functions, set the __HEAPSIZE__ macro within nc_define.inc to 0. This inhibits the unnecessary libraries from being linked, helping to save the ROM size. To use a C++ program, do not comment out this initialization part because the malloc function is used as a runtime library.

(3) Precautions to take when writing an original initialization program

To add your own initialization program in the assembler startup program, pay attention to the following:

(1) If the U or B flags are altered in your initialization program, restore these flags to their previous state on exit from the initialization program. Also, do not change the content of the SB register.

(2) To call subroutines written in C from your initialization program, pay attention to the following two points:

- The B and D flags must be cleared before you call.
- The U flag must be set before you call.

(4) Initializing global class objects and calling the termination function

When you compile and link C++ sources that use global class objects, a link error may occur, generating a message to the effect that no C$INIT section can be found. In that case, you need to call the function to initialize global class objects, or __CALL_INIT, immediately before you call the main function. Furthermore, it is necessary to call the function to terminate global class objects, or __CALL_END, immediately after you call the main function.

This part in ncrt0.a30 is commented out, so remove the comment to make this part compiled, as necessary.

d. Setting the size of the stack section

The stack section includes an area used for the user stack and an area used for the interrupt stack.
Set the interrupt stack size in the symbol __ISTACKSIZE__ in nc_define.inc.
Also, if the user stack needs to be used separately from the interrupt stack, set the user stack size in the symbol __STACKSIZE__ in nc_define.inc.

e. Setting the size of the heap area

When you use the heap area, please set the size of the symbol __HEAPSIZE__ needed.
Be sure that the heap area does not exceed the physical RAM area when you set its size.
To use a C++ program, reserve storage for the heap area as necessary, because the malloc function is used as a runtime library.
To link C++ program object files whose default attribute of RAM data pointer is "near," it is necessary that the heap area be located in an area with the near attribute.

```
__HEAPSIZE__        .equ        0300H       ; HEEP SIZE definition
```

Figure 2.19 Example of Setting the size of the heap area (nc_define.inc)

f. Setting the interrupt vector table

Use the -start linkage option to set the address of the vector section. The INTB register is initialized to the address where the vector section starts.

g. Setting the processor mode register

In a part of ncrt0.a30 shown in Figure 2.20, set the processor operation mode at address 04H (processor mode register) that is appropriate for the system in which your program will be incorporated.

```
;---------------------------------------------------------------------
; after reset,this program will start
;---------------------------------------------------------------------
                :
            (Omitted)
                :
            mov.b       #00h,04h            ;set processer mode
                :
            (Omitted)
                :
```

Figure 2.20 Example of Setting the Processor Mode Register (ncrt0.a30)

For details about the processor mode register, see the user's manual of your microprocessor.

2.2.3 Customizing Memory Mapping

a. Structure of sections

For compilers in a native environment, the executable files generated by the compiler have their addresses mapped to memory by the operating system such as UNIX. However, for compilers in a cross environment like this compiler. the user must determine the memory mapping.

This compiler maps programs and data to the microcontroller memory in units of "sections," separately for each program functionality such as the storage class of variables, variables with initial values, variables without initial values, string data, interrupt handling routines, interrupt vector tables, etc.

The section name representing each section consists of a section base name and its attribute, as shown in Figure 2.21.

Section base name ___ Attribute

Figure 2.21 Section Name

Table 2.13 lists the section base names. Table 2.14 lists the section attributes.

Table 2.13 Section Base Names

Section Base Names	Content
data	Stores data that has initial values.
bss	Stores data that does not have initial values.
rom	Stores character strings and the data specified by const qualifier.

Table 2.14 Section Attributes

Attribute	Meaning		Applicable section base name
I	Section to hold initial values of data		data
N/F/S	N	near attribute[4]	data, bss, rom
	F	far attribute	
	S	SBDATA attribute	data, bss
E/O	E	Even data size	data, bss, rom
	O	Odd data size	

Table 2.15 lists the contents of sections other than those based on the naming rules described above.

[4] Note that near and far are NC30-specific qualifiers. Use of these qualifiers makes it possible to specify addressing modes explicitly.
near … Accessible in the range of from address 00000H to address 00FFFFH
far ….. Accessible in the range of from address 00000H to address 0FFFFFH

Table 2.15 Section Names

Section name	Content
fvector	Stores the contents of the microprocessor's fixed vector
heap_NE	This is a memory area dynamically allocated during program execution by memory management functions (malloc, new). This section can be located in any RAM area of the microprocessor. To link C++ program object files whose default attribute of RAM data pointer is "near," it is necessary that the heap area be located in an area with the near attribute.
program	Stores a program.
program_S	Stores a program specified by #pragma SPECIAL.
stack	This is the area used as stack. Referenced by the user stack pointer register (USP). Locate this section at addresses from 0400H to 0FFFFH.
istack	This is the area used as stack. Referenced by the interrupt stack pointer register (ISP). Locate this section at addresses from 0400H to 0FFFFH.
switch_table	Stores a jump table for switch statements. This section is generated only when the compile option "-fswitch_other_section (-fSOS)" is used.
vector	Stores the content of the microprocessor's interrupt vector table. The interrupt vector table can be mapped to any location of the microprocessor's entire memory space by INTB register relative addressing. For details, see the user's manual of your microprocessor.
C$INIT	Stores the addresses of constructors and destructors invoked for global class objects. This section must be located in the ROM.
C$VTBL	Stores the data needed to call virtual functions, if any present in class declaration. This section must be located in the ROM.
interrupt	This is a startup section that includes the entry symbol (start) defined in ncrt0.a30. Mapping this section that includes the entry symbol to a location preceding the program section, it is possible to have the optimizing linkage editor optimized effectively.

Specify mapping of these sections with the -start option at link time. An example of section mapping is shown in Figure 2.22

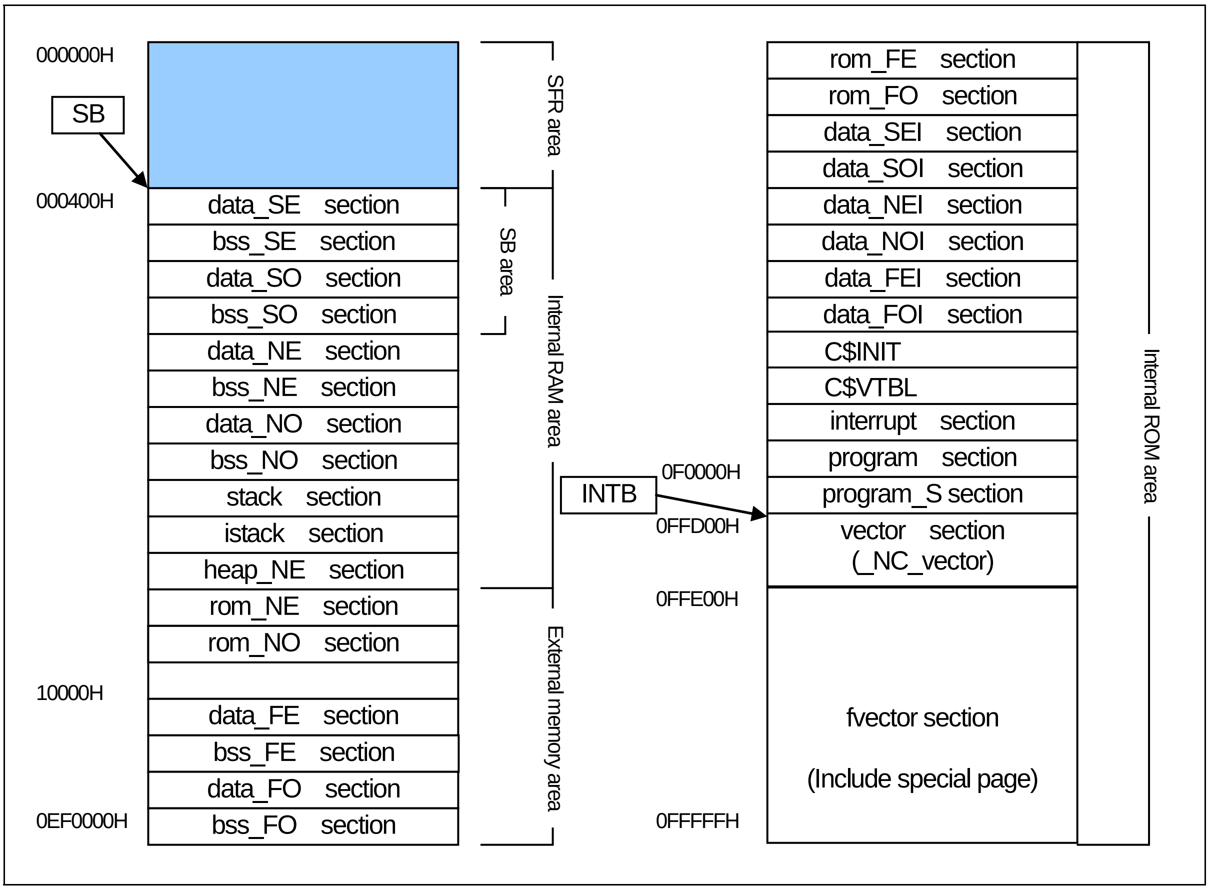


Figure 2.22 Example Section Mapping

(1) Rules for mapping sections to memory

Since sections are affected by the microprocessor's memory attributes (RAM or ROM), some sections can only be mapped to specific areas. Follow the rules described below when mapping sections to memory.

(1) Sections mapped to the RAM area

- stack section
- heap_NE section
- data_SE section
- data_SO section
- data_NE section
- data_NO section
- bss_SE section
- bss_SO section
- bss_NE section
- bss_NO section
- bss_FE section
- bss_FO section

(2) Sections mapped to ROM

- program section
- interrupt section
- fvector section
- rom_NE section
- rom_NO section
- rom_FE section
- rom_FO section
- data_SEI section
- data_SOI section
- data_NEI section
- data_NOI section
- data_FEI section
- data_FOI section
- C$INIT section
- C$VTBL section

Note also that some sections can only be mapped to specific areas in the microprocessor's memory space.

(1) Sections that can only be mapped to OH–0FFFFH (near area)

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

(2) Sections that can only be mapped to 0F0000H–0FFFFFH

- program_S section

(3) Sections that can be mapped to the entire memory space of the M16C/60 series

- programsection
- vector section
- data_NEI section
- data_NOI section
- data_FE section
- data_FO section
- data_FEI section
- data_FOI section
- data_SEI section
- data_SOI section
- bss_FE section
- bss_FO section
- rom_FE section
- rom_FO section
- C$INIT section
- C$VTBL section

If any of the following data-related sections have a size of 0, they do not always need to be defined.

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

b. Setting up the interrupt vector table

For programs that use interrupt processing, it is necessary to set the addresses of interrupt functions in the interrupt vector table (section "vector"). If interrupt functions are defined using #pragma INTERRUPT that makes use of vector numbers or #pragma INTCALL, the linker generates the section "vector." However, if this definition is made using #pragma INTERRUPT that does not make use of vector numbers, the interrupt vector table needs to be set in sect30.inc.
The content of the vector table varies with each microprocessor type, and must therefore be set up to suit the type of the microprocessor used. For details, see the user's manual of your microprocessor.

(1) Setting up the interrupt vector table in sect30.inc

For programs that use interrupt processing, alter the interrupt vector table in the vector section of sect30.inc. Figure 2.23 shows an example interrupt vector table.

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

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

* dummy_int is a dummy interrupt processing function.
```

Figure 2.23 Example of Setting Up the Interrupt Vector Table

Follow the procedure described below to alter the interrupt vector table in the vector section of sect30.inc.

(1) Declare the interrupt processing functions as externally referenced by using the assembler directive .GLB.

(2) The registered label names for the interrupt processing functions created by this compiler have an underscore (_) added in front of the function name. Therefore, the interrupt function names declared here must be preceded by an underscore.

(3) Change function names from a dummy interrupt function name dummy_int in the appropriate interrupt vector table to the interrupt processing function name used.

Figure 2.24 shows an example of setting the UART1 transmission interrupt processing function uarttrn.

```
.lword    dummy_int       ; uart0 receive (for user)
.glb      _uarttrn                                      ← Process (1) above
.lword    _uarttrn        ; uart1 transmit (for user)   ← Process (2) above

(Omitted)
```

Figure 2.24 Example of Setting Up the Interrupt Vector Table

2.3 Preparing the C Startup Program

For programs written in C to be 'burned' into ROM, a startup program written in assembly language or C to initialize the microcontroller, locate sections, and set up interrupt vector tables is required. The startup program needs to be modified to suit the MCU type you're using and the system in which it is used.
This section describes a C startup program written in C and how to customize it.
Note that when, after launching the integrated development environment (High-performance Embedded Workshop), you select Application for the project type in creating a new project, a template for C startup programs is automatically generated in a folder . Modify this template to suit your need.

2.3.1 Generated Files

The C startup program has the following files.

(1) resetprg.c
Initializes the microprocessor.

(2) initsct.c
Initializes each section (by clearing to zeros and transferring their initial values).

(3) heap.c
Reserves storage for the heap area.

(4) fvector.c
Defines the fixed vector table.

(5) intprg.c
Declares the entry function of fixed vector interrupts.

(6) firm.c/firm_ram.c (Need not be altered)
Reserves storage for the program and workspace areas as dummy that are used by firm of the FoUSB/E8 when OnChipDebugger is selected.

(7) cstartdef.h
Defines each define value such as stack size and heap size.

(8) initsct.h (Need not be altered)
A file in which the process (assembler macro) to initialize each section is written.

(9) resetprg.h
Includes each header file.

(10) typedefine.h (Need not be altered)
Declares each type with typedef.

(11) sfrXX.h,sfrXX.inc
Registers the sfr definition header file in the workspace corresponding to the CPU that was selected when a project was created.

2.3.2 Processing in Each Generated File

- resetprg.c (Mandatory)

The content of this file varies with the selected MCU (M16C or R8C).

```
#pragma section program interrupt   -------------------------------------------------------------------   (1)

void start(void)                    -------------------------------------------------------------------   (2)
{
        _isp_     = &_istack_top;       // set interrupt stack pointer      ------------   (3)
        protect = 0x02U;                // change protect mode register ----------   (4)
        pmode0    = 0x00U;              // set processor mode register  ----------   (5)
        protect = 0x00U;                // change protect mode register ----------   (6)
        _flg_     = __F_value__;        // set flag register                ----------   (7)
        _sp_      = &_stack_top;        // set user stack pointer          ------------   (8)
        _sb_      = 0x400U;  // 400H fixation (Do not change)               ------------   (9)

        // set variable vector's address
        _asm("     ldc        #((topof vector)>>16)&0FFFFh,INTBH");   ------------   (10)
        _asm("     ldc        #(topof vector)&0FFFFh,INTBL");

        initsct();              // initlalize each sections                 -------------   (11)
#if __HEAPSIZE__ != 0
        heap_init();            // initialize heap                          ------------   (12)
#endif
#if __STANDARD_IO__ != 0
        _init();                        // initialize standard I/O          --------------   (13)
#endif
        _fb_ = 0U;  // initialize FB registe for debugger
//      _CALL_INIT();           // Remove the comment when you use global class object
        main();                         // call main routine               --------------   (14)

        _exit();                // call exit
}
```

(1) Maps the start function to the interrupt section.
(2) Declares the body of the CPU initialization function, start().
(3) Initializes the interrupt stack pointer.
(4) Sets the protect registers to be "write-enabled."
(5) Sets the processor mode register to "single-chip mode."
To change the mode, you need to alter this expression.
(6) Sets the protect registers to be "write-protected."
(7) Sets the U flag.
If "Use User Stack" is selected in the workspace creation wizard, the user stack pointer is set.
(8) If "Use User Stack" is selected in the workspace creation wizard, the user stack pointer is set.
(9) Sets the SB register to the address 0x400 (to set the start address of RAM).
(10) Sets the fixed vector address in the INTB register.
(11) Initializes each section (by clearing to zeros and transferring their initial values).
(12) Initializes the heap area.
To use the memory management functions, you need to enable a call to this function.
(13) Initializes the standard I/O functions.
To use the standard I/O functions, you need to enable a call to this function.
(14) Calls the main function.

- initsct.c (Mandatory)

The content of this file varies with the selected MCU (M16C or R8C).

```
void initsct(void)
{
        sclear("bss_SE","data","align");      ---------------      (1)
        sclear("bss_SO","data","noalign");
        sclear("bss_NE","data","align");
        sclear("bss_NO","data","noalign");

        sclear_f("bss_FE","data","align");    ---------------      (2)
        sclear_f("bss_FO","data","noalign");

        // add new sections refer to the above - mentioned.

        scopy("data_SE","data","align");      ---------------      (3)
        scopy("data_SO","data","noalign");
        scopy("data_NE","data","align");
        scopy("data_NO","data","noalign");

        scopy_f("data_FE","data","align");    ---------------      (4)
        scopy_f("data_FO","data","noalign");
}
```

(1) sclear Clears the bss sections in the near area to zeros.
If any bss section name has been changed or added using the #pragma SECTION bss function, it is necessary to alter or add NE and NO as a set.

sclear("section name_NE", "data.align");
sclear("section name_NO", "data,noalign");

Example: #pragma section bss If a section is added in bss2,
add the following to initsct.c:

sclear("bss2_NE", "data,align");
sclear("bss2_NO", "data,noalign");

(2) sclear_f Clears the bss sections in the far area to zeros.

(3) scopy Transfers initial values to the data sections in the near area.
If any data section name has been changed or added using the #pragma SECTION data function, it is necessary to change or add NE and NO as a set.

scopy("section name_NE","data,align");
scopy("section name_NO","data,noalign");

Example: #pragma section data If a section is added in data2,
add the following to initsct.c:

scopy("data2_NE","data,align");
scopy("data2_NO","data,noalign");

(4) scopy_f Transfers initial values to the data sections in the far area.

- heap.c (Needed only when you use memory management functions such as malloc)

```
#pragma SECTION      bss       heap       --------------    (1)

_UBYTE heap_area[__HEAPSIZE__];           --------------    (2)
```

(1) Maps the heap area to the heap_NE section.
* If the heap is odd bytes in size, it is mapped to the heap_NO section.

(2) Reserves storage for the heap area as many bytes as defined by __HEAPSIZE__.

- fvector.c (Mandatory)

```
#pragma sectaddress   fvector,ROMDATA      0xffdc    -----------    (1)

/////////////////////////////////////////////////////////////////////

#pragma interrupt/v _dummy_int    //udi              -----------    (2)
#pragma interrupt/v _dummy_int    //over_flow
#pragma interrupt/v _dummy_int    //brki
#pragma interrupt/v _dummy_int    //address_match
#pragma interrupt/v _dummy_int    //single_step
#pragma interrupt/v _dummy_int    //wdt
#pragma interrupt/v _dummy_int    //reserved
#pragma interrupt/v _dummy_int    //reserved
#pragma interrupt/v start                            -----------    (3)
```

(1) Outputs the sections and addresses of the fixed vector table.
* This program is used for startup purposes, and cannot therefore be used normally.

(2) Pads all fixed vectors but reset with dummy functions (_dummy_int).
The "#pragma interrupt/v function name" registers a function name in the vector. To write the function body, define it using #pragma interrupt separately from this declaration.

(3) Defines the entry function.
This registers the function executed upon reset in a fixed vector.

- intprg.c (For each MCU type, as needed)

```
// DMA0 (software int 8)
#pragma interrupt       _dma0(vect=8)
void _dma0(void){}

// DMA1 (software int 9)
#pragma interrupt       _dma1(vect=9)
void _dma1(void){}

// DMA2 (software int 10)
#pragma interrupt       _dma2(vect=10)
void _dma2(void){}

// DMA3 (software int 11)
#pragma interrupt       _dma3(vect=11)
void _dma3(void){}

  (Omitted)
```

(1) Declares variable vector interrupt functions.
This declares the functions corresponding to each variable vector interrupt. At the same time, it generates a variable vector table.

(2) Defines variable vector interrupt functions.
Write a process in each function corresponding to the interrupt vector number you use.

Example: To use interrupt vector No. 9 (DMA1)

```
#pragma interrupt       _dma1(vect=9)
void _dma1(void)
{
        //Write a process
}
```

(3) If intprg.c is unnecessary
Remove from registration in the file, so that it will be deselected from being linked.

- firm.c/firm_ram.c (OnChipDebugger, when FoUSB/E8 is selected)

Do not alter the content of this file.

The content of this file is altered depending on the selected MCU and whether FoUSB or E8 is selected.

```
#ifdef __E8__          // for E8                          ------------   (1)

#pragma section bss FirmRam                               -----------    (2)

#ifndef __WORK_RAM__
#define __WORK_RAM__          0x80
#endif

_UBYTE _workram[__WORK_RAM__];                            ------------   (3)

#pragma section bss FirmArea                              ------------   (4)
_far _UBYTE _firmarea[0x800];    // dummy for monitor     ------------   (5)

#else       // for FoUSB

#pragma section bss FirmRam                               ------------   (6)
_UBYTE _workram[0x80];           // for Firmware's workram   -----------   (7)

#pragma section bss FirmArea                              -----------    (8)
_far _UBYTE _firmarea[0x600];    // dummy for monitor     -----------    (9)
#endif
```

(1) Enables E8 when it is used.
(2) Reserves storage for the work ram area used by the firmware of E8 in the FirmRam_NE section.
(3) Reserves storage for the work ram area as many bytes as defined by __WORK_RAM__.
(4) Maps the firmware program of E8 to the FirmArea section.
(5) Specifies the size of the firmware program.
(6) Reserves storage for the work ram area used by the firmware of FoUSB in the FirmRam_NE section.
(7) Reserves 0x80 bytes of storage for the work ram area. (It varies with the MCU type concerned).
(8) Maps the firmware program of FoUSB to the FirmArea section.
(9) Specifies the size of the firmware program.

- cstartdef.h (Mandatory)

```
#define __STACKSIZE__          0x80   ------------   (1)
#define __ISTACKSIZE__         0x80   ------------   (2)
#define __HEAPSIZE__           0x80   ------------   (3)
#define __STANDARD_IO__        0      ------------   (4)
#define __WATCH_DOG__          0      ------------   (5)
```

(1) Varies according to the stack size that was input in the workspace creation wizard.
(2) Varies according to the interrupt stack size that was input in the workspace creation wizard.
(3) Varies according to the heap size that was input in the workspace creation wizard.
(4) Set to 1 when "Use Standard I/O Functions" was selected in the workspace creation wizard.
(5) If the WATCH DOG feature needs to be enabled immediately after reset, set this item to 1. (For only the R8C family)

To change the above again after creating a new workspace, alter the relevant items directly in this file.

- initsct.h (Mandatory)

Do not change the content of this file

- resetprg.h (Mandatory)

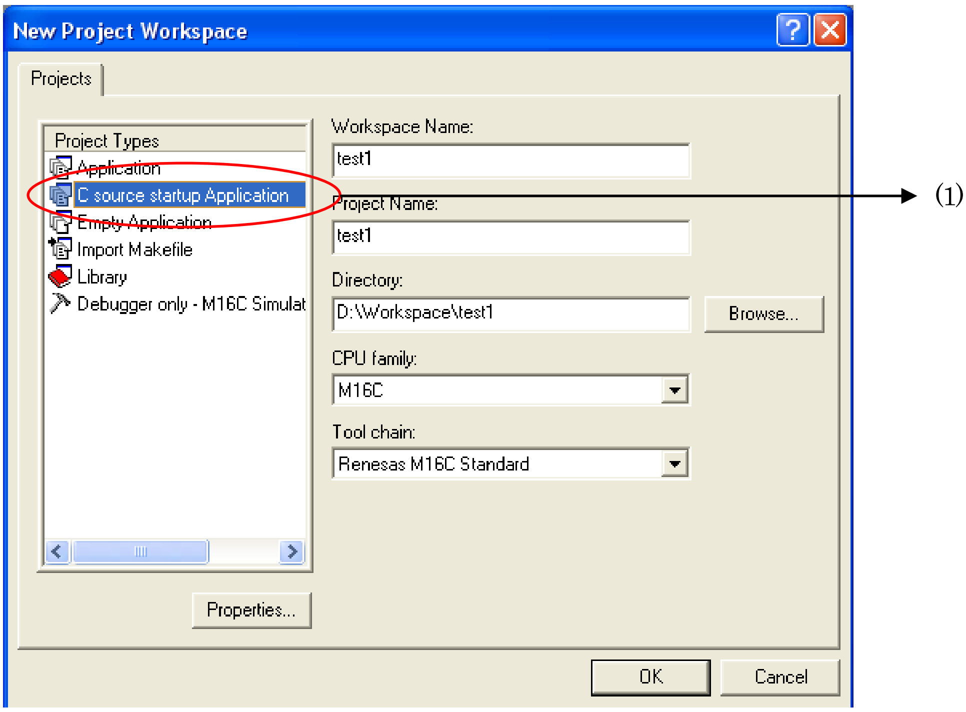
When you're using the on-chip debugger, see L1.3, "Regarding the FirmRam_NE Section and SB Register Value when On-Chip Debugger is Selected."

- typedefine.h (Mandatory)

Do not change the content of this file

2.3.3 Method for Generating C Startup

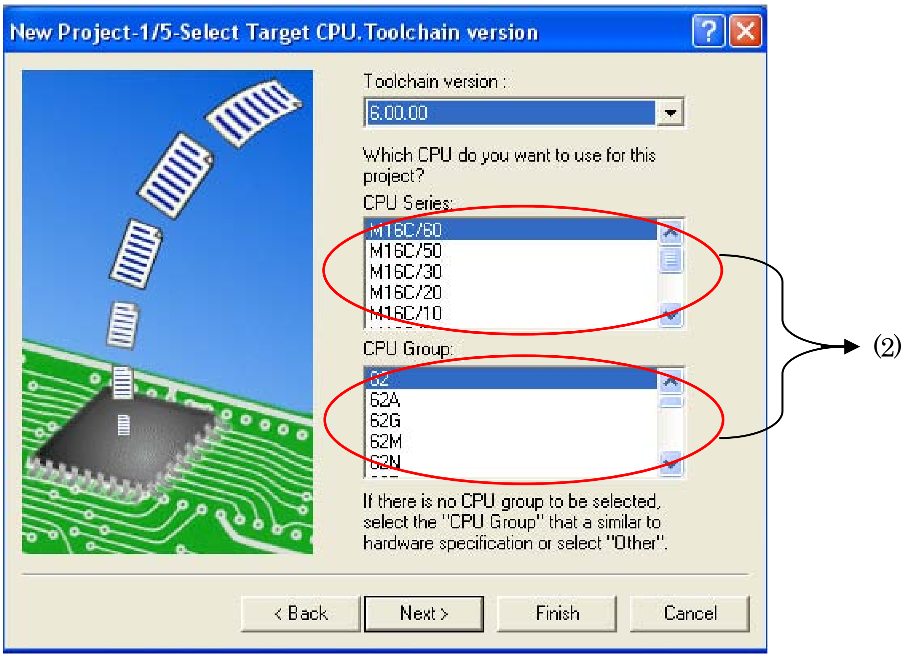
- Selecting a project that uses C startup



(1) Select C Source Startup Application in the left-side pane of the window.

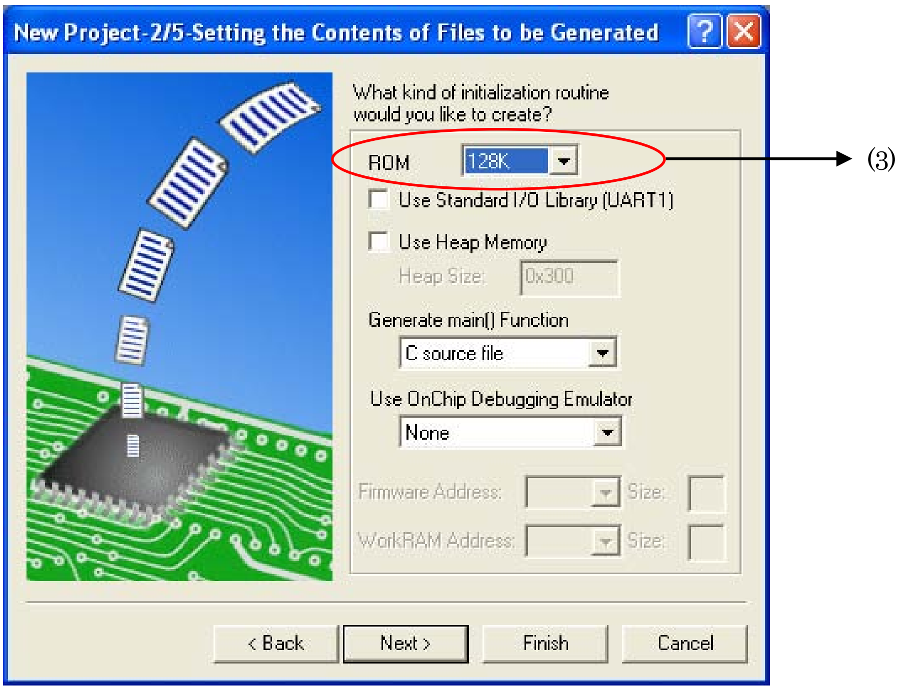
* If, while you have multiple C compilers installed, you select another microprocessor for the CPU type after selecting C Source Startup Application, you'll see the focus for C Source Startup Application moved to Application, with the result that C source startup is deselected. In that case, select C Source Startup Application again.

- Selecting the microprocessor type



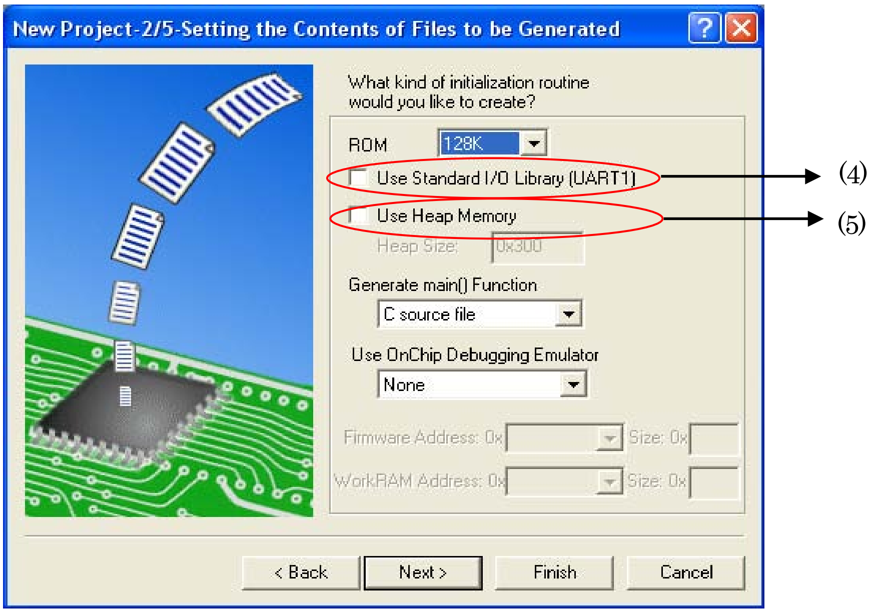
(2) Select the microprocessor type from CPU Series and CPU Group.
When selected, the corresponding sfr header file is registered in the workspace.
Also, a variable vector entry function (intprg.c) is registered.

- Selecting the ROM size



(3) The ROM size you select here has such an effect that sections with ROM attribute are mapped to memory appropriately when linked, according to the selected ROM size, as well as settings made at on-chip debugger selection.

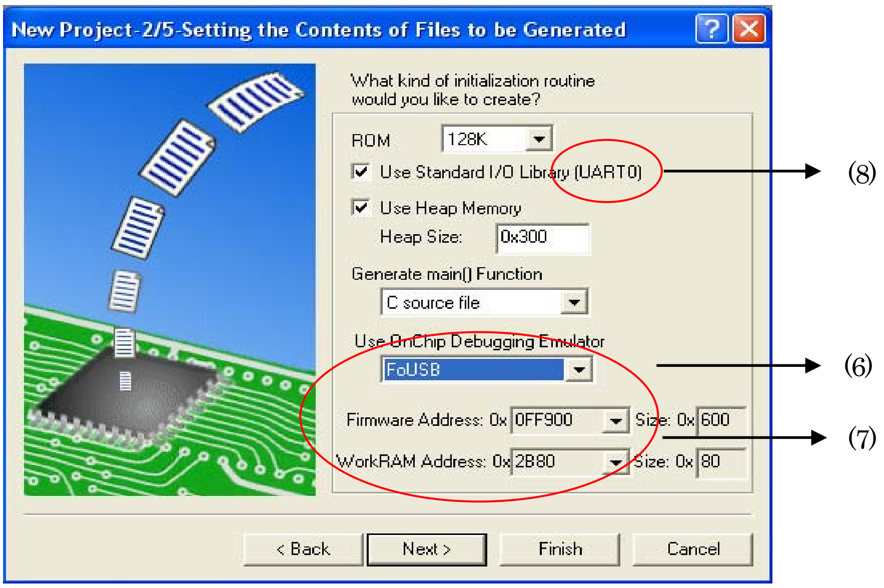
- Settings to make when using the standard I/O function library and memory management function library



(4) To use the standard I/O function library, check this checkbox.
When it is checked, calls to _init() in resetprg.c are enabled. Also, device.c and init.c are registered in the project.

(5) To use the memory management function library, check this checkbox.
When it is checked, calls to heap_init() in resetprg.c are enabled. Also, heapdef.h and heap.c is registered in the project.

- Selecting OnChipDebugger



(6) To use OnChipDebugger, select one from the dropdown list.
The selectable debuggers are FoUSB and E8.
However, either one or both of them cannot be selected depending on the microprocessor type you've selected.
Upon this selection , firm.c is registered, and bytes of memory occupied by the debugger, the one displayed in (7), is reserved as a variable area. That way, overlapping of memory spaces with the user program is avoided.

(7) Setting up FirmwareAddress and workRamAddress

Set up the areas occupied by FoUSB/E8 which consist of the program area for firmware and the RAM area for work. The settings here can only be changed when addresses are changeable by the debugger used.

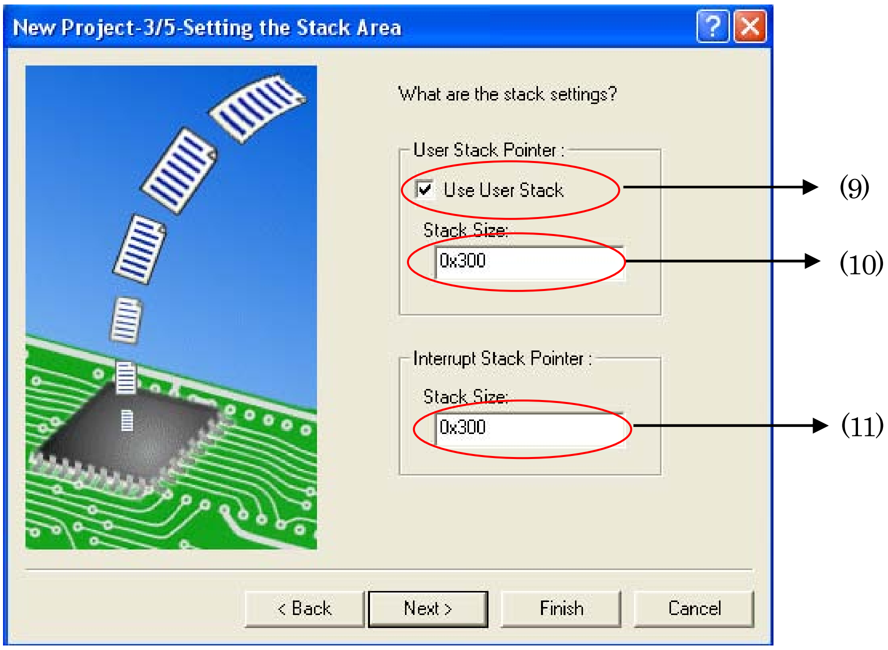
If these addresses have been changed when using the debugger, change this dialog box according to the settings made when using the debugger.

For the information on each address and size needed when you make a change, see the user's manual of your debugger.

(8) If you select OnChipDebugger while standard I/O function library is selected, (UART1) displayed in this dialog will change to (UART0).

This means that the standard I/O side is changed to UART0 because the standard I/O functions and If (UART0) is displayed here, set the compile option -D__UART0__ to perform conditional compilation.

- Selecting the stack size



(9) Choose whether the user stack is used.

If this checkbox is unchecked, the user stack is set 'not to be used' in the start function.

(10) Set the user stack size.

Alter the define value in cstartdef.h.

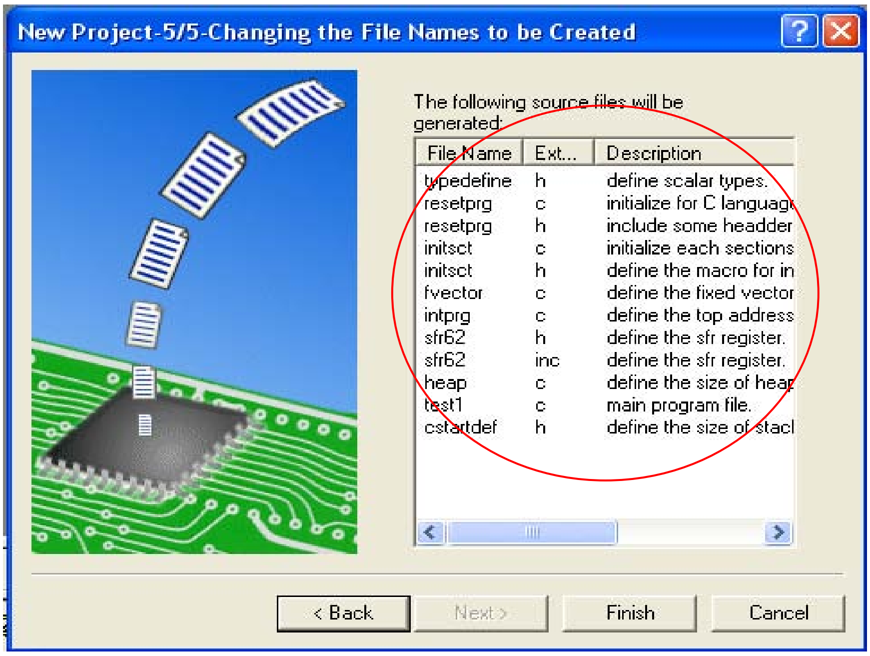
(11) Set the user stack size.

Alter the define value in cstartdef.h.

To change the stack size and HEAP size after creating a project, alter the respective values given below when you make settings in cstartdef.h.

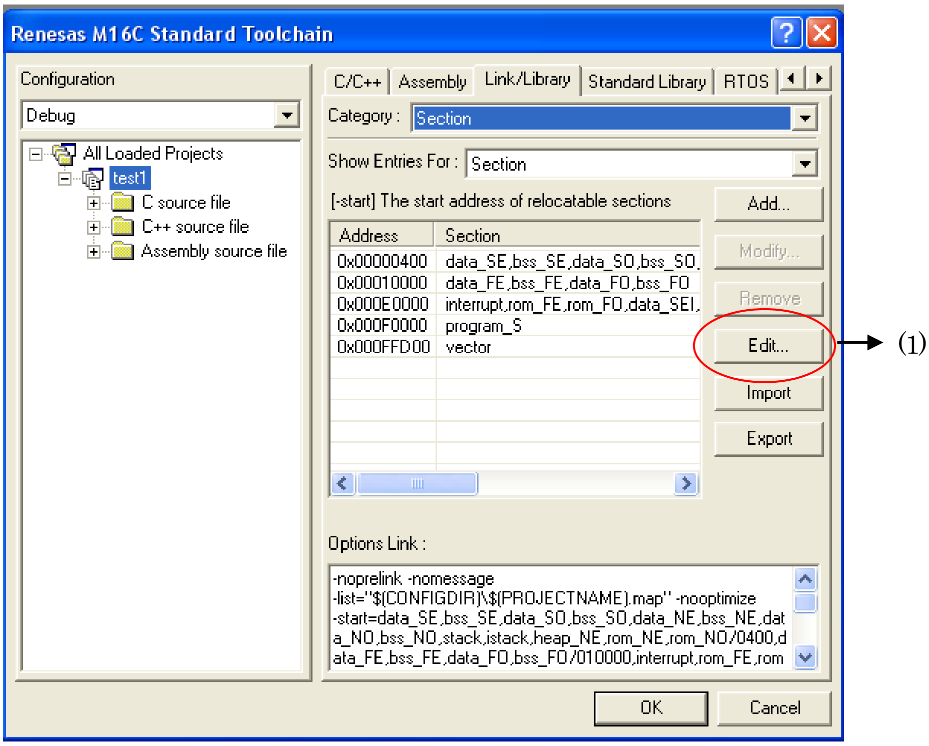
```
#define __STACKSIZE__      0x80
#define __ISTACKSIZE__     0x80
#define __HEAPSIZE__       0x80
```

- Registered file list

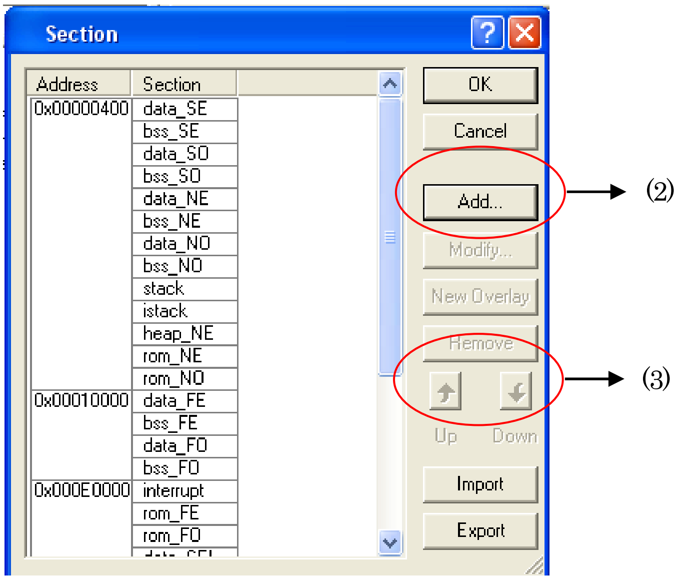


This list allows you to confirm the registered files.

- Order of sections



To check the order in which each section is linked and their link addresses, select Renesas M16C Standard Toolchain Link and look at Category: Section.



If you've added a new section with #pragma SECTION, for example, select the Edit button in (1) and open the window "Section Settings." While the focus is on Section, select the Add button in (2).

The window "Add Sections" will appear, so enter a new section name in it.
As the section you've entered is registered, use the UP/DOWN button in (3) to move the section to the area in which you want it located.

Chapter 3 Programming Technique

This chapter describes precautions to be observed when programming with the C compiler, NC30.

3.1 Notes

Renesas Electronics Corporation are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Electronics Corporation, Renesas Solutions Corp., or an authorized Renesas Semiconductor product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus orsystems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.

3.1.1 Notes about Version-up of compiler

The machine language instructions (assembly language) generated by this compiler vary with the startup options specified at compile time, version changes, etc. Therefore, if you've changed the startup options or upgraded the compiler version, please be sure to re-evaluate the behavior in whole of the program you created.
Furthermore, when the same RAM data is referenced (and its contents changed) between interrupt handling and non-interrupt handling routines or between tasks under realtime OS, always be sure to use exclusive control such as volatile specification. Also, use exclusive control for bit field structures which have different member names but are mapped into the same RAM.

3.1.2 Notes about the M16C's Type Dependent Part

When writing to or reading a register in the SFR area, it may sometimes be necessary to use a specific instruction. Because this specific instruction varies with each type of MCU, consult the user's manual of your MCU for details.
This compiler may generate instructions that cannot be used to write to or read from the registers in the SFR area. If an access to the SFR area is attempted as in a C program fragment in Figure 3.1, because the compiler generates instructions unusable in the SFR area, the interrupt request bit may not be determined correctly, causing an unintended behavior to occur.
To write to or read from the registers in the SFR area, write instructions directly in the program using asm functions. In this case, be sure to check that the generated code has no problems, regardless of which version of the compiler is used and whether options are specified or not.

```
#pragma ADDRESS   TA0IC    0055h          /* M16C/60 MCU's Timer A0 interrupt control register */

struct {
          char        ILVL : 3;
          char        IR : 1;                /* An interrupt request bit */
          char        dmy : 4;
} TA0IC;

void      wait_until_IR_is_ON(void)
{
          while(TA0IC.IR == 0)               /* Waits for TA0IC.IR to become 1 */
          {
                      ;
          }
          TA0IC.IR = 0;                      /* Returns 0 to TA0IC.IR when it becomes 1 */
}
```

Figure3.1 Example of a Program Fragment Written for the SFR Area

3.1.3 About Optimization

a. Regular optimization

The following are always optimized regardless of whether optimization options are specified or not.

(1) Meaningless variable access

For example, the variable port shown below does not use the readout results, so that readout operations are deleted.

```
extern int  port;

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

Figure3.2 Example of a Meaningless Variable Access (Optimized)

Although the intended operation in this example is only to read out port, the readout code actually is not optimized before being output. To suppress optimization, add the volatile qualifier as shown in Figure3.3

```
extern int volatile     port;

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

Figure3.3 Example of a Meaningless Variable Access (Optimization Suppressed)

(2) Meaningless comparison

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

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

Figure3.4 meaningless Comparison

In the example here, the variable 'c' is written to be char, so that this compiler handles it as unsigned char type. Since the values representable by unsigned char type range from 0 to 255, the variable 'c' will in no case have the value –1. Therefore, this compiler does not generate assembly language code for logically meaningless statements like the one shown here.

(3) Programs not executed

No assembly language code is generated for a program that will not logically be executed.

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

            i = 10;               ← Fragment not executed
}
```

Figure3.5 Program Not Executed

(4) Operation between constants

Operation between constants is performed when compiling.

```
int         func(void)
{
            int         i = 1 + 2;          ← Operation on this part is performed when compiling

            return i;
}
```

Figure3.6 Program Not Executed

(5) Selection of optimum instructions

Selection of optimum instructions as when using the STZ instruction or outputting shift instructions for division/multiplications, is always performed regardless of whether optimization options are specified or not.

b. About the volatile qualifier

Use of the volatile qualifier helps to prevent the referencing of variables, the order in which they are referenced, the number of times they are referenced, etc. from being affected by optimization.
However, avoid writing statements like those shown below which will be interpreted ambiguously.

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

a = b = c;              /* Whether a = c or a = b ? */
```

Figure3.7 Example of Ambiguously Interpreted volatile qualifier

3.1.4 Precautions on Using register Variables

a. register qualification and compile option "-fenable_register(-fER)"

If the compile option "-fenable_register(-fER)" is specified, the variables that are register-qualified so as to satisfy specific conditions can be forcibly assigned to registers. This facility is provided for improving generated codes without relying on optimization.
This feature is provided for improving the generated code without relying on optimization. Extensive use of this feature may result in poor efficiency, so be sure to check the generated code before making use of it.

b. About register qualification and optimization options

When optimization options are specified, the compiler assigns variables to registers as one functionality of optimization. This assignment feature is unaffected by whether the variables are register-qualified.

3.2 For Greater Code Efficiency

3.2.1 Programming Techniques for Greater Code Efficiency

a. Regarding Integers and Variables

(1) Unless required, use unsigned integers. If there is no sign specifier for int, short, or long types, they are processed as signed integers. Unless required, add the 'unsigned' sign specifier for operations on integers with these data types.[1]

(2) If possible, do not use >= or ← for comparing signed variables. Use != and = = for conditional judgments.

b. far type array

The far type array is referenced differently at machine language level depending on its size.

(1) When the array size is within 64K bytes
Subscripts are calculated with unsigned 16-bit integers. This ensures efficient access for arrays of 64K bytes or less in size.

(2) When the array size is greater than 64K bytes or unknown
Subscripts are calculated in 32-bit width.

Therefore, if it is known that the size will not exceed 64K bytes, code efficiency can be improved by writing a size expressly in the extern declaration of a far-type array shown in Figure3.8 or using the option "-fsmall_array(-fSA)"[2] to compile.

```
extern int far      array[];      ← Size is unknown, so subscripts are calculated as 32-bit values.
extern int far      array[10];    ← Size is within 64KB, so access is more efficient.
```

Figure3.8 Example extern-Declaration of far Array

c. Making the most of prototype declarations

This compiler makes efficient function calls possible by declaring prototypes for the functions concerned. This means that unless a function has its prototypes declared, this compiler places arguments to it on the stack following the rules in Table 3.1and passes the stack of arguments to the function when calling it

Table 3.1 Rules for Using Stack for Parameters

Data type(s)	Stacking rules
char signed char	Extended to int type when stacked
float	Extended to double type when stacked
otherwise	Not type extended when stacked

Therefore, unless function prototypes are declared, redundant type extensions may result.
Function prototype declarations help to suppress these redundant type extensions, as well as enable efficient function calls because they make it possible to assign arguments to registers.

[1] If there is no sign specifier for char-type or bitfield structure members, they are processed as unsigned.
[2] When the compile option "-fsmall_array (-fSA)" is specified, the compiler assumes an array of an unknown size to be within 64K bytes as it generates code. In the entry version, this option cannot be specified.

d. Using SB Register Efficiently

Use of the SB register[3] -based addressing mode helps to reduce the application program size (ROM capacity). This compiler permits declaration of the variables that use the SB register-based addressing mode by writing the program fragment shown in Figure3.9.

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

Figure3.9 Example of variable declaration using SB-based addressing mode

e. Other methods

In addition to the above,the ROM capacity can be compressed by changing program description s as shown below.

(1) Chabge a relatively small function that is called only once to an inline function.
(2) Replace an if-else statement with a switch statement. (This is effective unless the variable concerned is a simple variable such as an array,pointer,or structure.)
(3) For bit comparison, use '&' or '|' in place of '&&' or '||'.
(4) For a function which returns a value in only the range of char type, declare its return value type with char.
(5) For variables used overlapping a function call, do not use a register variable.
(6) When the compiler and the assembler set -goptimize in the option, the optimum jump instruction is selected.

3.2.2 Speeding Up Startup Processing

The ncrt0.a30 startup program includes routines for clearing the bss area. This routine ensures that variables that are not initialized have an initial value of 0, as per the C language specifications.
For example, the code shown in Figure3.10 does not initialize the variable, which must therefore be initialized to 0 (by clearing the bss[4] area) during the startup routine.

```
static int    i;
```

Figure3.10 Example Declaration of Variable Without Initial Value

Some applications do not require clearing the variables without initial values to 0. In such a case, comment out the bss area clearing part of the startup program, which will help to expedite the startup processing.

[3] In this compiler, SB register is initialized after reset. Later, is assumed to use fixed.
[4] The external variables in RAM which do not have initial values are referred to as "bss".

```
=====================================================================
; NEAR area initialize.
;--------------------------------------------------------------------
; bss zero clear
;--------------------------------------------------------------------
;   N_BZERO   (topof bss_SE),bss_SE
;   N_BZERO   (topof bss_SO),bss_SO
;   N_BZERO   (topof bss_NE),bss_NE
;   N_BZERO   (topof bss_NO),bss_NO
            :
    (omitted)
            :
=====================================================================
; FAR area initialize.
;--------------------------------------------------------------------
; bss zero clear
;--------------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
;   BZERO   (topof bss_FE),bss_FE
;   BZERO   (topof bss_FO),bss_FO
.endif
```

Figure3.11 Commenting Out Routine to Clear bss Area

3.3 Linking Assembly Language Programs with C Programs

3.3.1 Calling Assembler Functions from C Programs

a. Calling Assembler Functions

To call assembler functions from a C/C++ program, do this call by an assembler function name in the same way as for calls to functions written in C/C++.
The beginning label names of assembler functions must have an underscore (_) added at the top of the name. To call assembler functions from a C/C++ program, use the name of the assembler function (beginning label name) that has had its underscore removed. To call assembler functions from a C program, always write a prototype declaration for the assembler function, as shown in Figure3.12.
Figure3.12 shows an example of how to write a program fragment to call the assembler function asm_func from a C program.

```
extern void  asm_func( void );          ← Assembler function prototype declaration

void         main()
{
                :
             (omitted)
                :
             asm_func();                ← Calls assembler function
}
```

Figure3.12 Example of Calling Assembler Function Without Parameters(sample.c)

```
             .glb        _main
_main:
                :
             (omitted)
                :
             jsr         _asm_func      ← Calls assembler function(preceded by '_')
             rts
```

Figure3.13 Compiled result of sample.c(sample.a30)

To call assembler functions from a C++ program, always add "extern C" to the assembler function prototype, as shown in Figure 3.14. This inhibits the name qualification inherent in C++.
Figure3.14 shows an example of how to write a program fragment to call the assembler function asm_func from a C++ program.

```
extern "C" {
      extern void asm_func(void);
}

void   main()
{
      (omitted)
          :
       asm_func();
}
```

Figure3.14 Example of a Program Fragment to Call the Assembler Function asm_func from a C++ Program

b. When assigning arguments to assembler functions

When passing arguments to assembler functions, use the extended function "#pragma PARAMETER". This #pragma PARAMETER passes arguments to assembler functions via 32-bit general-purpose registers (R2R0, R3R1), 16-bit general-purpose registers (R0, R1, R2, R3), or 8-bit general-purpose registers (R0L, R0H, R1L, R1H) and address registers(A0, A1).

The following shows the sequence of operations for calling an assembler function using #pragma PARAMETER:

(1) Declare with #pragma PARAMETER the register name that is used for the argument list for the assembler function.

(2) After writing a #pragma PARAMETER declaration, declare the prototype for the assembler function.
(Only when compiled as a C program, it is possible to declare function prototypes before a #pragma PARAMETER declaration.)

Figure3.15 is an example of using #pragma PARAMETER when calling the assembler function asm_func.

```
#pragma PARAMETER asm_func(R0, R1)
extern unsigned int asm_func(unsigned int, unsigned int);     ← Parameters are passed via the R0 and R1
                                                                registers to the assembler function.
void          main(void)
{
              int           i = 0x02;
              int           j = 0x05;

              asm_func(i, j);
}
```

Figure3.15 Example of Calling Assembler Function With Parameters (sample2.c)

```
            .SECTION  program,CODE,ALIGN
            ._file    'sample.c'
            ._line    5
;## # C_SRC :         {
            .glb      _main
_main:
            enter     #04H
            ._line    6
;## # C_SRC :                   int         i = 0x02;
            mov.w     #0002H,-2 [FB]          ;   i
            ._line    7
;## # C_SRC :                   int         j = 0x05;
            mov.w     #0005H,-4 FB]           ;   j
            ._line    9
;## # C_SRC :                   asm_func(i, j);
            mov.w     -4 [FB],R1  ;   j                 ← Parameters are passed via the R0 and R1
            mov.w     -2 [FB],R0  ;   i                   registers to the assembler function.
            jsr       _asm_func
            ._line    10
;## # C_SRC :         }
            exitd
E1:
            .align
            .glb      _asm_func         ← Calls assembler function(preceded by '_')
            .END                          As for the output assembler name of the function specified by
                                          #pragma PARAMETER, the _(underscore) is added always previously.
```

Figure3.16 Compiled result of sample2.c(sample2.a30)

c. Limits on Parameters in #pragma PARAMETER Declaration

The following parameter types cannot be declared in a #pragma PARAMETER declaration.

- structure types and union type parameters
- 64bit integer type (long long) parameters
- floating point type (double) or long double type parameters

Furthermore, return values of structure or union types cannot be defined as the return values of assembler functions.

3.3.2 Writing Assembler Functions

a. Method for writing the called assembler functions

The procedure for writing processing at the entry and exit to and from an assembler function is described below.

(1) Specify section names using the assembler pseudo-command .SECTION.
(2) Global specify function name labels using the assembler pseudo-command .GLB.
(3) Add the underscore (_) to the function name to write it as label.
(4) To alter the B and U flags in the function, save the flag register to the stack.
(5) If you modified the B and U flags within the function, restore the flag register from the stack.
(6) Write the RTS instruction.

To rewrite the contents of SB and FB registers, save the registers to the stack on entry to the function and restore them from the stack on exit from the function. However, before rewriting the SB and FB registers, make sure that no adverse effects are incurred in the entire path from entry to exit of the assembler function by rewriting of these registers.

Figure3.17 is an example of how to code an assembler function. In this example, the section name is program, which is the same as the section name output by NC30.

```
            .section    program, align          ← (1)
            .glb        _asm_func, SYM1         ← (2)
_asm_func:                                      ← (3)
            pushc       FLG                     ← (4)
            mov.w       SYM1, R1
            mov.w       SYM1+2,R3
            popc        FLG                     ← (5)
            rts                                 ← (6)
            .END
```

Figure3.17 Example Coding of Assembler Function

b. Returning Return Values from Assembler Functions

When returning values from an assembler function to a C language program, registers can be used through which to return the values for the integer, pointer, and floating- point types. Table 3.2 lists the rules on calls regarding return values. Figure3.18 shows an example of how to write an assembler function to return a value.

Table 3.2 Calling Rules for Return Values

Return value type	Rules
_Bool type char type	R0L register
int type near pointer type	R0 register
float type long type far pointer type	The 16 low-order bits are stored in the R0 register and the 16 high-order bits are stored in the R2 register as the value is returned.
double type long double type	The value is stored in 16 bits each beginning with the MSB in order of registers R3, R2, R1, and R0 as it is returned.
long long type	The value is stored in 16 bits each beginning with the MSB in order of registers R3, R1, R2 and R0 as it is returned.
Structure type Union type Class type	Immediately before calling the function, the far address indicating the area for storing the return value is pushed to the stack. Before the return to the calling program, the called function writes the return value to the area indicated by the far address pushed to the stack.

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

Figure3.18 Example of Coding Assembler Function to Return long-type Return Value

c. Referencing C/C++ Variables

Because assembler functions are written in different files from the C/C++ program, only the C/C++ global variables can be referenced.
When including the names of C/C++ variables in an assembler function, precede them with an underscore (_). Also, in assembler language programs, external variables must be declared using the assembler pseudo instruction .GLB.
Figure3.19 is an example of referencing the C program's global variable counter from the assembler function asm_func.

```
C program:

unsigned int        counter;          ← C program global variable

void        main(void)
{
               :
            (omitted)
               :
}


Assembler function:

            .glb        _counter      ← External declaration of C program's global variable
_asm_func:
               :
            (omitted)
               :
            mov.w       _counter, R0  ← Reference
```

Figure3.19 Referencing a C Global Variable

d. Precautions to take when writing interrupt handling with assembler functions

If you are writing a program (function) for interrupt processing, the following processing must be performed at the entry and exit.

(1) Save the registers (R0, R1, R2, R3, A0, A1 and FB) at the entry point.
(2) Restore the registers (R0, R1, R2, R3, A0, A1 and FB) at the exit point.
(3) Use the REIT instruction to return from the function.

Figure3.20 is an example of coding an assembler function for interrupt processing.

```
            .section    program
            .glb        _func
_int_func:
            pushm       R0,R1,R2,R3,A0,A1,FB     ← Save registers
            mov.b       #01H, R0L
               :
            (omitted)
               :
            popm        R0,R1,R2,R3,A0,A1,FB     ← Pull registers
            reit                                 ← Return to C program
            .END
```

Figure3.20 Example Coding of Interrupt Processing Assembler Function

e. Precautions to take when calling C/C++ functions from the assembler

Note the following when calling a function written in C from an assembly language program.

(1) Call the C/C++ function using a label preceded by the underscore (_) or the dollar ($).
(2) The C/C++ functions do not save the register contents as of the time they are called. To call C/C++ functions from an assembly-language program, save the data and address registers before the call.
(3) For C++ functions, declare "extern C."

```
extern "C" void foo( void )
{
        ..............
}
```

3.3.3 Precautions to Take when Writing Assembler Functions

When writing the assembly-language functions (subroutines) called from C/C++ functions, pay attention to the following.

a. Notes on Handling B and U flags

When returning from an assembler function to a C/C++ language program, always make sure that the B and U flags are in the same condition as they were when the function was called.

b. Notes on Handling FB Register

If the value of the FB (frame base register) is altered in an assembler function, control may become unable to return normally to the C/C++ program from which the function was called. If alterations are unavoidable for reasons of system design, save the FB value to the stack at the top of the function and restore it on return.

c. Notes on Handling General-purpose and Address Registers

The general-purpose registers (R0, R1, R2, R3) and address registers (A0, A1) can have their contents modified in assembler functions without a problem.

d. Passing Parameters to an Assembler Function

Use the #pragma PARAMETER function if you need to pass parameters to a function written in assembly language. The parameters are passed via registers.
Figure3.21 shows the format (asm_func in the figure is the name of an assembler function).

```
#pragma PARAMETER asm_func(R0, R1)
unsigned int near asm_func(unsigned int, unsigned int);        ← Prototype declaration of assembler function
```

Figure3.21 Prototype declaration of assembler function

#pragma PARAMETER passes arguments to assembler functions via the 16-bit general-purpose registers (R0, R1, R2, or R3), 8-bit general-purpose registers (R0L, R0H, R1L, or R1H), and address registers (A0 or A1). Also, the 16-bit general-purpose registers and address registers are combined to configure a 32-bit register (R3R1, R2R0, or A1A0), via which to pass arguments to assembler functions. Note that #pragma PARAMETER needs to be declared before an assembler function prototype declaration. (When not compiled as a C program, a prototype for any assembler function that is declared before a #pragma PARAMETER declaration has no effect.)

However, types of the following arguments cannot be declared in a #pragma PARAMETER declaration.

- structure types and union type parameters
- 64bit integer type (long long) parameters
- floating point type (double) or long double type parameters

You also cannot declare the functions returning structure or union types as the function's return values.

3.4 Other

3.4.1 Precautions on Transporting between NC-Series Compilers

NC30 basically is compatible with Renesas C compilers "NCxx" at the language specification level (including extended functions). However, there are some differences between the compiler (this manual) and other NC-series compilers as described below.

a. Difference in default near/far

The default " near/far" in the NC series are shown in Table 3.3. Therefore, when transporting the compiler (this manual) to other NC-series compilers, the near/far specification needs to be adjusted.

Table 3.3 Default near/far in the NC Series

Compiler	RAM data	ROM data	Program
NC308	near (However, pointer type is far)	far	far Fixed
NC30	near	far	far Fixed
NC30 (R8C)	Near Fixed	Near Fixed	far Fixed
NC30 (R8CE)	near	far	far Fixed
NC79	near	near	far
NC77	near	near	far

Appendix A Command Option Reference

This appendix describes how to start the compile driver of this compiler and the functionality of its startup options. The description of startup options here also includes those of the assembler and the linkage editor that can be started from this compiler.

A.1 Compile Driver Input Format

```
%nc30△[startup option]△<[ assembler source file name]△
                        [object file name]△[C/C++ source file name]>

%:     Prompt
< >:   Mandatory item
[ ]:   Optional item
△:     Space
```

Figure A.1 Compile Driver's Input Format

```
% nc30   -osample   sample.c<RET>

<RET>: Return key.
```

Figure A.2 Compile Driver's Command Input Example

A.2 Startup Options

A.2.1 Options for Controlling the Compile Driver

-c

Compile driver control

Function: Creates an object file (extension ".obj") and finishes processing.

Notes: When this option is selected, no absolute files are generated.

-D *identifier*

Compile driver control

Function: The function is the same as the preprocess command #define.
Multiple identifiers can be specified.

Supplement: Shown below is an example where multiple identifiers mac1 and mac2 are specified in the -D option.
% nc30 -Dmac1=1 -Dmac2=2 sample.c<RET>
%:Denotes the prompt.
<RET>: Denotes the return key.

Syntax: nc30Δ-D identifier[=constant]△<C/C++ source file>
* [= constant] is optional.

Notes: The number of identifiers that can be defined may be limited by the maximum number of characters that can be specified on the command line of the operating system of the host machine.

-dsource -dS

Comment option

Function: Generates an assembler source file (extension ".a30") (not removed even after assembling).

Supplement: Do not assemble the assembler source files generated by this option.

-dsource_in_list -dSL

List file option

Function: Generates an assembler list file (extension ".lst").

-E

Compile driver control

Function: Processes only preprocess commands and outputs the result to standard output.

Notes: When this option is selected, assembler source files (extension ".a30"), object files (extension ".obj"), absolute files (extension ".abs"), etc. are not generated.

-I *directory name*

Compile driver control

Function: Specifies the directory name in which to search for files to be referenced by the preprocess command #include.

Supplement: An example of specifying two directories (dir1 and dir2) for the "-I" option is shown below.
% nc30 -Idir1 -Idir2 sample.c<RET>
%: Indicates the prompt.
<RET>: Indicates the Return key.

Syntax: nc30△-I directory name△<C/C++ source file>

Notes: The number of directories that can be defined may be limited by the maximum number of characters that can be specified on the command line of the operating system of the host machine.

-P

Compile driver control

Function: Invokes only preprocess commands, creates a file (extension .i) and stops processing.

Notes:
(1) When this option is selected, no assembler source files (extension .a30), object files (extension .obj), absolute files (extension .abs), etc are generated.
(2) The file (extension .i) generated by this option does not include the #line command generated by the preprocessor. To get a result that includes #line, select the -E option and redirect.

-S

Compile driver control

Function: Creates an assembler source files (extension .a30). No object files are generated.

Notes: When this option is selected, object files (extension ".obj"), absolute files (extension ".abs"), etc. are not generated.
Template functions are output as static functions in the .a30 file.
Be aware that if the assembler source files generated by this option are assembled, C/C++ level debug information is lost.

-silent

Compile driver control

Function: Suppresses the display of copyright notices at startup.

-U *predefined macro*

Compile driver control

Function: Undefines predefined macro constants.

Syntax: nc30△-U predefined macro△<C/C++ source file>

Notes: This option allows you to undefine the NC30 and M16C predefined macros.

-lang

Compile driver control

Function: Specifies the source file language.
When the -lang=c option is specified, the input file is compiled as a C (C89) source file.
When the -lang=cpp option is specified, the input file is compiled as a C++ source file
When the -lang=ecpp option is specified, the input file is compiled as an Embedded C++ source file.
If, while this option is omitted, the filename extension is .ccp, .cc, or .cp, the input file is compiled as a C++ source file. If the filename extension is .c, the input file is compiled as a C (C89) source file. If the source file has the extension ".a30," it is handled as an assembler source file, even when it is specified otherwise by this option.

Syntax: nc30△-lang=c△<C/C++ source file>
nc30△-lang=cpp△<C/C++ source file>
nc30△-lang=ecpp△<C/C++ source file>

Notes: Embedded C++ specifications do not support catch, const_cast, dynamic_cast, explicit, mutable, namespace, reinterpret_cast, static_cast, template, throw, try, typeid, typename, using, multiple inheritance, and virtual base class. If any of these items is written in the source file, error messages are output.
When using the EC++ library, be sure to specify the -lang=ecpp option.

-preinclude

Compile driver control

Function: Function: Includes the content of a specified file at the top of the compilation unit. If there are multiple file names, they can be specified by separating each with a comma (,). If there are multiple folders in which the option is specified, the folders are searched in the order they are specified, from left to right.

Syntax: nc30△-preinclude=<filename>[, • • •]△<C/C++ source file>

Notes: If this option is specified multiple times, all of the specified files are included.

-exception, -noexception

Compile driver control

Function: When the -exception option is specified, the C++ exception handling facilities (try, catch, and throw) are enabled.
When the -noexception option is specified, the C++ exception handling facilities (try, catch, and throw) are disabled.
Note that when the -exception option is specified, code performance may be reduced.
If this option is omitted, -noexception is assumed by default.

Syntax: nc30△-exception△<C/C++ source file>
nc30△-noexception△<C/C++ source file>

Notes: To enable the exception handling facility between files, observe the following:

- Do not specify the -noprelink option in the optimizing linkage editor.

The -exception option can only be specified when compiling C++. If specification of -lang=cpp is nonexistent and the input file extension is .c or .i, the -exception option cannot be specified. Neglect of this restriction will incur a warning.

-rtti

Compile driver control

Function: Specifies that runtime type identification be enabled or disabled.
When -rtti=on is specified, dynamic_cast and typeid are enabled.
When -rtti=off is specified, dynamic_cast and typeid are disabled.
If this option is omitted, -rtti=off is assumed by default.

Syntax: nc30△-rtti=on△<C/C++ source file>
nc30△-rtti=off△<C/C++ source file>

Notes: Do not register in a library the object files (.obj) that were created after specifying this option or output them in relocatable form (.rel) in the optimizing linkage editor. Such an act may result in a duplicate symbol error or an undefined symbol error.
Note that -rtti=on can only be specified when compiling C++. If specification of -lang=cpp is nonexistent and the input file extension is .c or .i, the -rtti=on cannot be specified. Neglect of this restriction will incur a warning.

A.2.2 Options Specifying Output Files

-dir *directory name*

Output file specification

Function: This option allows you to specify an output destination directory for the output file.

Syntax: nc30△-dir directory-name△<C/C++ source file>

Notes: The source file information used for debugging is generated starting from the directory from which the compiler was invoked (the current directory).
Therefore, if output files were generated in different directories, the debugger, etc. must be notified of the directory from which the compiler was invoked.

-o *file name*

Output file specification

Function: Specifies the file generated by optlnk. A path name that includes a directory name can also be specified. Always be sure that the file extension is omitted.
If both -dir and -o are specified and the specified -o includes a directory, files are output to the path specified by -o, no matter what directory is specified by -dir.

Syntax: nc30△-o file name△<C/C++ source file>

A.2.3 Version Information and Command Line Display Options

-v

Display command program name

Function: Displays the name of the command program that is being executed internally while compiling files.

Notes: Use lowercase v for this option.

-V

Display version information

Function: Displays the version information of each command program executed internally by the compiler, then finishes processing.

Supplement: Use this option to check whether the compiler has been installed correctly. The correct version numbers of commands executed internally by the compiler are listed in Release Notes.
If the version numbers in Release Notes do not match those displayed using this option, the compiler may not have been installed correctly.

Notes:
(1) Use uppercase V for this option.
(2) If this option is selected, all other options have no effect.

A.2.4 Options for Debugging

-g

Output debug information

Function: Outputs debug information to object files.

Notes: When debugging your program at the C/C++ language level, always specify this option.
Specification of this option does not affect code generated by the compiler.
When the -finfo option is specified, -g also becomes effective.
When -fSB_auto(-fSBA) option is specified, -g is enabled also.

-genter

Output enter instruction

Function: Always outputs the enter instruction when calling a function.

Notes:

(1) When using the debugger's stack trace facility, always specify this option. Without this option, the correct result cannot be obtained.

(2) When this option is selected, the compiler generates code to construct the stack frame using the enter instruction at entry of the function regardless of whether it is necessary. Consequently, the ROM size and the amount of stack used may increase.

-gno_reg

Suppress debug information for register variables

Function: Suppresses output of debug information for register variables.

Notes: Use this option to suppress the output of debug information for register variables if that information is unnecessary. This will help to speed up downloading to the debugger.

A.2.5 Optimization Options

The effects of main optimization options are listed in Table A.1.

Table A.1 Effects of Optimization Options

Effect	-O	-OR	-OS	-OSA	-OSFA
Speed	Good	Bad	Good	Good	Good
ROM size	Good	Good	Bad	Good	Same Note
Stack used	Good	Bad	Same	Bad	Bad

Good: Improved (or the same as you do not use the option)
Bad: Worsened (or the same as you do not use the option)
Same: Not changed
Note: If there are many functions that do not have the stack frame, the code size will increase.

-O[1-5]

Optimization

Function:

Performs optimization that is effective for both speed and ROM size. This option can be specified simultaneously with the -g option. Unless a number (level) is specified, -O3 is assumed.

-O1: Performs optimization in the manner described below.

- Assign variables to registers.
- Delete meaningless conditional expressions.
- Delete statements that are not logically executed.

-O2: Same as -O1.

-O3: Executes the following optimization addition to the one performed by -O1.

- Putting bit manipulations together.
- Constant folding of floating-point numbers.
- Inline padding of standard library functions.

-O4: Executes the following optimization addition to the one performed by -O3.

- Replace references to the variables declared by the const qualifier with constants.

-O5: Executes the following optimization addition to the one performed by -O4.

- Optimize address computations of pointers, structures, etc. (if the option -OR is concurrently specified).
- Strengthen optimization on pointers (if the option -OS is concurrently specified).

However, the compiler may not be able to output normal code when the following conditions are met.

- Different variables point to the same memory location at the same time.
- Those variables are used in one and the same function.

Example:

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

void        test1(void)
{
            int         b;
            *p = 9;
            a = 10;
            b = *p;             /* Inadvertently replaces "*p" with "9" by optimization */
            printf( "b = %d (expect b = 10)¥n",b );
}
```

Execution result:

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

-O[1-5]

Optimization

Notes: The bit manipulating instructions (BTSTC and BTSTS) cannot be used to write and read to and from the registers in the SFR area.
This compiler, if an optimization option (-O5) is used, may generate bit-manipulating instructions (BTSTC, BTSTS) for assembly language code.
If, in a program written as in the example below, the input file is compiled using the optimization option (-O5), interrupt request bits may not be determined correctly, resulting in an unintended behavior.

```
C sources in which the optimization option must no be used:

#pragma  ADDRESS  TA0IC  0055h      /* M16C/62 timer A0 interrupt control register */
struct {
        char ILVL : 3;
        char IR : 1;                /* Interrupt request bit */
        char dmy : 4;
} TA0IC;

void    wait_until_IR_is_ON(void)
{
        while (TA0IC.IR == 0)       /* Waits until the bit is set to 1 */
        {
                ;
        }
        TA0IC.IR = 0;               /* Resets the bit to 0 when it is 1 */
}
```

If it is confirmed that the bit manipulating instructions (BTSTC and BTSTS) have been output for the SFR area, take one of the following corrective measures before compiling the source. In either case, be sure to confirm that the generated code has no problems.

- Use some other optimization option than "-O5."
- Use the asm function to write instructions directly in the program.
- Add the "-O5OA" option.

-OR

Optimization

Function: Performs ROM size-oriented optimization, in preference over speed. This option can be specified simultaneously with the -g and -O options.

Notes: When this option is used, the source line information may be partly changed in the course of optimization. For this reason, the program may appear to be acting differently when it is debugged.
If you do not want the source line information to be changed, use the -Ono_break_source_debug(-ONBSD) option to suppress optimization.

-OS

Optimization

Function: Performs speed-oriented optimization, in preference over the ROM size.
This option can be specified simultaneously with the -g and -O options.

-OR_MAX -ORM

Optimization

Function: Performs optimization that places priority on ROM size.

Explanation:

(1) The compile options listed below are enabled.
- -O5
- -OR
- -O5OA
- -goptimize
- -fchar_enumerator (-fCE)
- -fdouble_32 (-fD32)
- -fno_align (-fNA)
- -fno_carry (-fNC)
- -fsmall_array (-fSA)
- -fuse_DIV (-fUD)

(2) To select this option in the integrated development environment or High-performance Embedded Workshop, be sure to enable "Size and speed" on the Compiler tab of Renesas M16C Standard Toolchain and then select the checkbox "Perform ROM Size-Prioritized, Maximum Optimization."

Notes:

(1) The source line information may be partly changed in the course of optimization. For this reason, the program may appear to be acting differently when it is debugged. If you do not want the source line information to be changed, select the compile option -Ono_break_source_debug(-ONBSD) option to suppress optimization.

(2) Depending on the debugger used, the enum type may not be referenced correctly.

(3) When a function is defined or declared, it requires prototype declaration. If there is no prototype declaration, invalid code may be generated.

(4) Debug information for double type is handled as float type. In the C watch window and global window of the debugger or simulator, therefore, double type is displayed as float type.

(5) When using a far-type pointer to indirectly access memory dynamically allocated by the malloc function, etc. or ROM data mapped to the far area, be careful not to access the data overlapping the 64-Kbyte boundary.

(6) If this option is selected in combination with the compile option "-R8C" or "-R8CE," the functionality of the compile option "-fno_carry(-fNC)" is nullified.

(7) If a divide operation results in an overflow, a different behavior than stipulated in ASNI will result.

(8) If you specify this option, use the standard library generated by the library generator with -fno_align(-fNA) added.

-OS_MAX -OSM

Optimization

Function: Performs optimization that places priority on the number of cycles.

Explanation:

(1) The compile options listed below are enabled.

- -O4
- -OS
- -Oforward_function_to_inline(-OFFTI)
- -goptimize
- -Oloop_unroll=10 (-OLU=10)
- -Ostatic_to_inline (-OSTI)
- -Osp_adjust(-OSA)
- -fchar_enumerator (-fCE)
- -fdouble_32 (-fD32)
- -fno_carry (-fNC)
- -fsmall_array (-fSA)
- -fuse_DIV (-fUD)

(2) To select this option in the integrated development environment or High-performance Embedded Workshop, be sure to enable "Size and speed" on the Compiler tab of Renesas M16C Standard Toolchain and then select the checkbox "Perform Speed-Emphasized, Maximum Optimization."

Notes:

(1) The ROM size increases because for statements are unrolled.
(2) Assembly language code is generated for source lines in which the bodies of static functions that became to be handled as inline functions are written.
(3) To forcibly make any function to be handled as an inline function, declare it with the inline specifier.
(4) Depending on the debugger used, the enum type may not be referenced correctly.
(5) When a function is defined or declared, it requires prototype declaration. If there is no prototype declaration, invalid code may be generated.
(6) Debug information for double type is handled as float type. In the C watch window and global window of the debugger or simulator, therefore, double type is displayed as float type.
(7) When using a far-type pointer to indirectly access memory dynamically allocated by the malloc function, etc. or ROM data mapped to the far area, be careful not to access the data overlapping the 64-Kbyte boundary
(8) If this option is selected in combination with the compile option "-R8C" or "-R8CE," the functionality of the compile option "-fno_carry(-fNC)" is nullified.
(9) If a divide operation results in an overflow, a different behavior than stipulated in ASNI will result.
(10) Be sure that the bodies of inline functions are defined in the same file as the inline functions are declared.
(11) No structures and unions can be used for parameters to inline functions. If this is attempted, a compile error results.
(12) Inline functions cannot be called indirectly. If such an indirect call is written, a compile error will result.
(13) Inline functions cannot be called recursively. If such a recursive call is written, a compile error will result.

-Ocompare_byte_to_word -OCBTW

Optimization

Function: Performs bytewise comparison on contiguous areas in words.

Notes: This is effective only when the -O[1–5] (or -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)) option is selected.

-Oconst -OC

Optimization

Function: Performs optimization to replace references to the variables declared by the const qualifier with constants. This is effective when the option -O4 or greater is specified, too. However, storage for variables is reserved.

Supplement: This optimization is performed when the following conditions are met at the same time:

(1) Variables except bit fields and unions
(2) Variables for which the const qualifier is specified but are not specified to be volatile
(3) External variables whose initialization is written in the same C source file
(4) Variables that are initialized with constants or const-qualified variables

-Oforward_function_to_inline -OFFTI

Optimization

Function: Expands all inline functions in-line.

Supplement: Calls to inline functions require that before an line function can be called, its body must be defined. Use of this option, however, allows the body of an inline function to be defined after it is called.

Notes:

(1) Be sure that the bodies of inline functions are defined in the same file as these functions are declared.
(2) Structures and unions cannot be used for parameters to inline functions. If this restriction is neglected, a compile error result.
(3) Indirect calls to inline functions cannot be made. If such a call is written in the program, a compile error result.
(4) Recursive calls to inline functions cannot be made. If such a call is written in the program, a compile error result.
(5) To expand defined-in-class functions in-line, this option is required.

-Oloop_unroll[=*loop count*] -OLU[=*loop count*]

Unroll a loop

Function: Unrolls code as many times as the loop count without revolving the loop statement. The "loop count" can be omitted. When omitted, this option is applied to a loop statement with a maximum loop count of 5.

Supplement: Unrolled code is output for only the ‘for’ statement where the number of times it is executed is known. Specify the upper-limit count for which times the target for loop to be unrolled is revolved. By this default, this option is applied to the for statement where the loop is revolved up to 5 times.

Notes: The ROM size increases because for statements are unrolled.

-Ono_asmopt -ONA

Suppress assembler optimizer

Function: Suppresses optimizations by the assembler optimizer "aopt30."

-Ono_bit -ONB

Suppress optimization

Function: Suppresses the optimization that puts bit manipulations together.

Supplement: When the O[3–5], -OR,-OS, -OR_MAX(-ORM), or -OS_MAX(-OSM) option is selected, operations to assign constants to consecutive bit fields that are mapped to a memory area are put together into one operation for optimization.
Because such optimization is undesirable if successive bit manipulations have an order of operation to observe, as in I/O bit fields, use this option to suppress optimization.

-Ono_break_source_debug -ONBSD

Suppress optimization

Function: Suppresses the optimization that affects source line information.

Supplement: When the O[3–5], -OR, or -OR_MAX(-ORM) option is selected, the compiler may perform optimization that affects source line information. Use this option to suppress such optimization.

-Ono_float_const_fold -ONFCF

Suppress optimization

Function: Suppresses the constant-folding processing of floating-point numbers.

Supplement: This compiler performs, by default, constant-folding processing. Here is an example.

```
Before optimization:
        (val/1000e250)*50.0

After optimization:
        val/20e250
```

In this case, if the application uses the full dynamic range of floating-point numbers, the result of calculation may differ as the order of calculation is changed. This option suppresses the constant folding in floating-point representation, so that the order of calculation written in the C source is guaranteed.
The functionality of this option is effective only when the input file is compiled as a C program.

-Ono_logical_or_combine -ONLOC

Suppress optimization

Function: Suppresses the optimization that puts logical ORs together.

Supplement: If one of options -O3 or greater, -OR, or -OS is specified when compiling as in the example shown below, the compiler performs optimization that puts logical ORs together.

```
Example:
        if( a & 0x01 || a & 0x02 || a & 0x04 )
              |
              ↓       (Optimization)
        if( a & 0x07 )
```

In this case, the variable ‘a’ is referenced up to 3 times, but after optimization it is referenced only once.
However, if the variable ‘a’ has any significance in its references as in I/O, the program may not operate correctly. In such a case, select this option to suppress the optimization that puts logical ORs together.
Note, however, that if variables are declared as volatile, logical ORs are not combined for optimization.

-Ono_stdlib -ONS

Suppress optimization

Function: Suppresses optimization that embeds standard library functions in-line, modifies library functions, etc.

Supplement: This option suppresses the following optimization:

- Optimization for replacing the standard library functions such as strcpy() and memcpy() with the SMOVF instruction, etc.
- Optimization for changing the library functions to those appropriate for near/far attributes of parameters
- Optimization for changing mathematic function libraries when -fdouble_32(-fD32) is used.

Notes: When functions with the same names as the standard library functions are created on the user side, the need may arise to select this option.

-Osp_adjust -OSA

Combine stack correction code

Function: Performs optimization that puts stack correction codes after function calls together.

Supplement: Normally, each time a function is called, the stack pointer is corrected in order to free storage for parameters to the function. When this option is used, corrections of the stack pointer are performed collectively, rather than for each function call made.

Example:

In the following case, the stack pointer is corrected each time func1() and then func2() is called (i.e., corrected twice). When this option is used, the stack pointer is corrected only once.

```
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 );
}
```

Notes: The option -Osp_adjust helps to reduce the ROM size, as well as to speed up processing. However, the amount of stack used may increase.
To use this option, always specify one of -O[1–5], -OR, or -OS at the same time.

-Ostack_frame_align -OSFA

Align stack frame

Function: Aligns the frame stack on even address boundary.

Supplement: If auto variables that have an even size are mapped to odd addresses, memory access requires one more cycle than when they are mapped to even addresses. When this option is specified, the stack frame is aligned in such a way that even-size auto variables are mapped to even addresses, thereby speeding up memory access.

Notes:

(1) This alignment is not performed for the functions specified with the following #pragma directives:
- #pragma INTHANDLER
- #pragma HANDLER
- #pragma ALMHANDLER
- #pragma CYCHANDLER
- #pragma INTERRUPT[1]

(2) In the startup program, make sure the initial values of stack pointers are mapped to even addresses.

(3) Also, be sure that this option is applied for all programs you compile.

(4) If you specify this option, use the standard library that was generated by the library generator with this option added.

[1] The alignment described above is not performed for interrupt functions because the stack pointer value at the time an interrupt is generated is not guaranteed to be of an even number. For this reason, if this option is specified for any function called from an interrupt function, processing speed may be lowed down rather than speeded up.

-Ostatic_to_inline -OSTI

Handle stack functions as inline functions

Function: Handles the functions declared as static (i.e., static functions) as the functions declared as inline (i.e., inline functions), generating inline-expanded assembly language code.

Supplement: When the following conditions are met, static functions are handled as inline functions, generating inline-expanded assembly language.

(1) The function concerned is a static function whose body is written before a function call.
- A call to a function and the body of that function are written in the same source file.
- Ignore this condition if you've selected the "-Ofoward_function_to_inline" option.

(2) The static function concerned does not have its address obtained in the program.

(3) The static function concerned is not recursively called.

(4) In assembly language code outputs of the compiler, no frame (storage reserved for auto variables, etc) is constructed.
- Whether a frame is constructed depends on the written content of the function concerned and the combined use of other optimization options.
- Ignore this condition if you've selected the "-Ofoward_function_to_inline" option.

Shown below is an example of writing a static function that is expanded in-line.

```
extern int        i;

static int func(void)
{
        return i++;
}

void    main(void)
{
        int       s;

        s = func();
        s = func();
}
```

The function func() is inline-expanded in the respective places where it is called within the function main().

Notes:

(1) Assembly language code is always generated for source lines in which the bodies of the static functions that became to be handled as inline functions are written.

(2) If some functions do not need to be forcibly handled as inline functions, declare them with the inline specifier.

(3) This option is required when intra-class defined functions need to be expanded in-line.

-O5OA

Suppress optimization

Function: Suppresses generation of code using the bit manipulating instructions (BTSTC and BTSTS) when the optimization option "-O5" is selected.

Notes: The bit manipulating instructions (BTSTC and BTSTS) cannot be used to write and read to and from the registers of the SFR area. If, when the optimization option "-O5" is selected, code is generated that uses the bit manipulating instructions for write and read to and from the registers of the SFR area, use this option to suppress code generation.

-goptimize

Function: Generates in the output file the additional information that is used at the time of intermodule optimization.
Files that have had this option specified become the subject of intermodule optimization when linked.
This option cannot be specified simultaneously with -fSB_auto.

A.2.6 Options for Modifying Generated Code

-fansi

Modify generated code

Function: Enables the options listed below when the input file is compiled as C++.

-fnot_reserve_far_and_near:	Does not handle far and near as the reserved words.

Enables the options listed below when the input file is compiled as C.

-fnot_reserve_asm:	Does not handle asm as the reserved word.
-fnot_reserve_far_and_near:	Does not handle far and near as the reserved words.
-fnot_reserve_inline:	Does not handle inline as the reserved word.
-fextend_to_int:	Promotes char-type data to type int when the data is operated on.

Supplement: When this option is selected, the compiler generates code in conformity with ANSI standards.
Since asm and inline are the keywords of standard C++, they are always handled as the keywords when compiling C++, regardless of whether this option is specified.
When compiling C++, integral promotions are applied to data when it is operated on, regardless of whether this option is specified.

-fchar_enumerator -fCE

Modify generated code

Function: Handles types of enumerator as unsigned char type, not as int type.

Notes: When this option is selected, some debuggers may not be able to refer to enum type correctly.

-fconst_not_ROM -fCNR

Modify generated code

Function: Does not handle types specified with the const qualifier as ROM data.

Supplement: The data specified with const are, by default, mapped to the ROM area.

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

In the above case, the array 'array' is mapped to the ROM area. By specifying this option, it is possible to locate the 'array' in the RAM area.
You do not normally need to use this option.

-fdouble_32 -fD32

Modify generated code

Function: Processes double type as float type.

Supplement:
(1) If you specify this option, be sure to declare a function prototype. If prototype declarations are nonexistent, invalid code may be generated.
(2) When this option is selected, the debug information for double type is handled as float type. In the C watch window and global window of the debugger or simulator, therefore, the information is displayed as float type.
(3) When you use this facility, be aware that float type and double type cannot be overloaded in C++ programs.
(4) When this option is added, -Wnon_prototype is enabled at the same time.
(5) Mathematical functions are replaced with single-precision mathematical functions.

-fenable_register -fER

Modify generated code

Function: Assigns variables that are specified as register-storage-class to registers.

Supplement: When optimization is performed for "assignments of auto variables to registers," it may not always be possible to obtain the optimum solution. This option is provided as a means of increasing the efficiency of optimization by instructing variable assignments to registers in a program under the above situation.
When this option is selected, the following register-specified variables are forcibly assigned to registers.

- Integral type variables
- Pointer variables

Notes: An excessive use of register specification may have an adverse effect that the efficiency decreases. Be sure to check the generated assembler source files before using this specification.

-fextend_to_int -fETI

Modify generated code

Function: Promotes char-type or signed char-type data to type int when the data is operated on (as stipulated in ANSI standards).

Supplement: In ANSI standards, char-type or signed char-type data to type int when the data is evaluated. This is because operations on char types (e.g., c1=c2*2/c3;) would otherwise cause the char type to overflow in the middle of operation, producing an unexpected result.

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

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

In this case, the char type overflows in the course of the operation "c2*2," so that the correct result may not be obtained. By selecting this option, it is possible to obtain the correct result.
The reason why promotions to type int are disabled by default is because it is conducive to increasing the ROM efficiency any further.
When compiled as C++ programs, integral promotions are applied to data when it is operated on, regardless of whether this option is specified.

-ffar_RAM -fFRAM

Modify generated code

Function: Changes the default attribute of RAM data to far.

Supplement: The RAM data (variables) are located in the near area by default. Use this option when you want RAM data to be located in other than the RAM area (64-Kbyte area).

Notes: This option cannot be used in combination with -R8C or -R8CE.

-finfo

Modify generated code

Function: Outputs the inspector information required for utl30 into the object file.

Notes:
(1) No check is made for the use of global variables in the asm function. In utl30 too, therefore, use of the asm function is ignored.
(2) -finfo includes -g.
(3) Even if this option is selected, the generated code of the compiler is unaffected.
(4) Do not use this option for the compiler sources that do not load the headers generated by utl30.

-fbit -fB

Modify generated code

Function: Generates code assuming that bitwise manipulating instructions can be executed using absolute addressing for all external variables mapped to the near area.

Supplement: If the 'near' external variables subject to bit manipulation are located in the M16C Series, R8C Family memory space 0000h through 1FFFh, specification of this option helps to increase the efficiency of codes generated by the compiler.
When, in single-chip applications, the RAM is located in the above memory space, specifying this option should prove effective. If an attempt is made to operate on variables that are located in any other memory space, an error will result when linking.

-fno_carry -fNC

Modify generated code

Function: Suppresses carry flag addition when data is indirectly accessed using far-type pointers.

Supplement: When accessing structures or 32-bit data indirectly using far-type pointers, this option generates code that does not perform carry addition to the 16 high-order bits of the far-type pointer (32-bit pointer), assuming that the data is not mapped across the 64-Kbyte boundary. As a result, increased efficiency can be expected.

Notes: When using far-type pointers to indirectly access memory dynamically allocated by the malloc function, etc. or ROM data mapped to the far area, be sure that the data is not accessed overlapping a 64-Kbyte boundary.
This option cannot be used in combination with -R8C or -R8CE.

-fauto_128 -fA1

Modify generated code

Function: Limits the size of the stack frame used to a maximum of 128 bytes. (The maximum size of stack frames is, by default, 255 bytes.)

-ffar_pointer -fFP

Modify generated code

Function: Changes the default attribute of pointer type to far.

Supplement:
(1) The pointer-type variables in this compiler have, by default, the near attribute. Use this option to change the default attribute to far.
(2) The pointer variables that are defined with the near qualifier are processed as having the near attribute, regardless of whether this option is specified.

Example:

```
char near *p;    // Processed as near pointer
```

-fnear_ROM -fNROM

Modify generated code

Function: Changes the default attribute of ROM data to near.

Supplement: The ROM data (e.g., const-specified variables) are, by default, located in the far area. By selecting this option, it is possible to locate ROM data in the near area.

-fno_align -fNA

Modify generated code

Function: Does not align the start addresses of functions.

Supplement: The output assembler is changed as follows:

- The assembler directive .align is not output in front of function symbols.
- align is not specified in the assembler directive .section for sections with the code attribute.

Notes: This use option for all programs you compile.
If you specify this option, use the standard library that was generated by the library generator with this option added.

-fno_even -fNE

Modify generated code

Function: When outputting data, does not separate odd and even data. This means that all data are mapped to the odd sections (data_NO, data_FO, data_NOI, data_FOI, bss_NO, bss_FO, rom_NO, and rom_FO).

Supplement: By default, the odd-size and the even-side data are output to separate sections.

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

In the above case, the variable 'c' and the variable 'i' are output to separate sections. This is to ensure that even-size variables 'i' are located at even addresses. As a result, fast access can be expected when accessing data in 16-bit bus width.
Use this option when the CPU is used in only 8-bit bus width and the number of sections needs to be reduced.

Notes: When #pragma SECTION is used to change the name of a section, data is mapped to the newly named section.

-fno_switch_table -fNST

Modify generated code

Function: For switch statements, generates code that performs comparison before a jump, instead of generating code that uses a jump table.

Supplement: If this option is not selected, the compiler generates code that uses a jump table only when the code size will become smaller than otherwise.

Notes: For large functions where the code size per function exceeds 32K bytes, if code is generated that uses a jump table for switch statements, a link error may occur. In that case, be sure to specify this option.

-fnot_address_volatile -fNAV

Modify generated code

Function: Does not handle the variables specified by #pragma ADDRESS as those specified to be volatile.

Supplement: If I/O variables are optimized in the same way as for variables in RAM, unexpected behavior may result. This can be avoided by specifying volatile for I/O variables.
Since "#pragma ADDRESS normally is used for I/O variables, they are processed assuming that they are of volatile property, without explicit volatile specification. This option suppresses such processing.

Notes: You do not normally need to use this option.

## -fnot_reserve_asm	-fNRA

Modify generated code

Function: Does not handle asm as a reserved word.

Supplement: The _asm that has the same functionality is handled as a reserved word.
When compiling the input file as a C++ program, the compiler always handles asm as a reserved word, regardless of whether this option is specified.

## -fnot_reserve_far_and_near	-fNRFAN

Modify generated code

Function: Does not handle far and near as reserved words.

Supplement: The _far and _near that have the same functionality are handled as reserved words.

## -fnot_reserve_inline	-fNRI

Modify generated code

Function: Does not handle inline as a reserved word.

Supplement: The _inline has the same functionality is handled as a reserved word.
When compiling the input file as a C++ program, the compiler always handles inline as a reserved word, regardless of whether this option is specified.

## -fsmall_array	-fSA

Modify generated code

Function: When referencing a far-type array whose total size is unknown, calculates subscripts in 16 bits assuming that the total size of the array is within 64 Kbytes.

Supplement: If when referencing the members of a far-type array the size of the array is unknown, the compiler, by default, calculates subscripts in 32 bits in case an array in size of 64 Kbytes or more has to be handled.

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

In the above case, because the total size of the 'array' array is unknown when compiled, the compiler calculates the subscript 'j' in 32 bits. When this option is selected, the compiler calculates the subscript 'j' in 16 bits assuming that the total size of the 'array' array is 64 Kbytes or less. This helps to increase the processing speed and reduce the code size.
Renesas recommends using this option whenever the size of one array does not exceed 64 Kbytes.

-fswitch_other_section -fSOS

Modify generated code

Function: Outputs a jump table for switch statements to some other section than the program section.

Supplement: The section name is "switch_table."

Notes: You do not normally need to use this option.

-fchange_bank_always -fCBA

Modify generated code

Function: Outputs code that switches the bank from one to another every time.

Supplement: Specify this option when you are using the #pragma EXT4MPTR or _ext4mptr feature and want to declare multiple instances of a pointer variable to 4-Mbyte space.

Notes: This option cannot be used in combination with -R8C or -R8CE.

-fauto_over_255 -fAO2

Modify generated code

Function: Changes the stack frame size per function that can be reserved to a maximum of 64 Kbytes. (The maximum size of stack frames is, by default, 255 bytes.)

Supplement: 1. This option cannot be used in combination with the #pragma SBDATA feature. When a file that contains a description of #pragma SBDATA is compiled, the warning shown below is output, with the description of #pragma SBDATA ignored.
compile option -fauto_over_255 is specified,#pragma SBDATA was ignored.
===>#pragma SBDATA xxx;
* This is because #pragma SBDATA cannot be used since the SB register is used to construct a stack frame.
2. Specify this option for the files described below.
a. When there is a function that requires a stack frame of 255 bytes or more (hereafter referred to as function A)
=====>Files in which function A is written
b. When an interrupt occurs while processing function A (hereafter referred to as interrupt A) and a variable declared by #pragma SBDATA is accessed from the interrupt A
=====>Files in which interrupt A is written

-fsizet_16 -fS16

Change the bit size of type definition

Function: Changes type definition size_t from type unsigned long to type unsigned int.

Supplement: When this option is selected, the libraries linked become as follows:

a) If a project is built in the integrated environment (High-performance Embedded Workshop), the library generator automatically generates a library and optlnk links that library.

b) If nc30.exe is invoked at the command prompt and executed until after a link process, nc30.exe automatically links the following libraries.

- When the -R8C option is used simultaneously r8cs16.lib
- When the -R8CE option is used simultaneously r8ces16.lib
- Otherwise nc30s16.lib

c) If optlnk is invoked at the command prompt separately, link the library generated by the library generator.

-fptrdifft_16 -fP16

Change the bit size of type definition

Function: Changes type definition ptrdiff_t from type signed long to type signed int.

Supplement: When this option is selected, the libraries linked become as follows:

a) If a project is built in the integrated environment (High-performance Embedded Workshop), the library generator automatically generates a library and optlnk links that library.

b) If nc30.exe is invoked at the command prompt and executed until after a link process, nc30.exe automatically links the following libraries.

- When the -R8C option is used simultaneously r8cs16.lib
- When the -R8CE option is used simultaneously r8ces16.lib
- Otherwise nc30s16.lib

c) If optlnk is invoked at the command prompt separately, link the library generated by the library generator.

-fuse_DIV -fUD

Modify generated code

Function: Changes generated code for divide operations.

Supplement: For divide operations where the dividend is a 4-byte value, the divisor is a 2-byte value, and the result is a 2-byte value or when the dividend is a 2-byte value, the divisor is a 1-byte value, and the result is a 1-byte value, the compiler generates microprocessor instructions "div.w(divu.w)" and "div.b(divu.b)."

Notes:

(1) If the divide operation results in an overflow when this option is selected, a different behavior than stipulated in ANSI will occur.

(2) The div instruction of the M16C Series, R8C Family has such a characteristic that when the operation resulted in an overflow, the result becomes indeterminate. Therefore, if the input file is compiled without selecting "-fuse_DIV(-fUD)," the compiler calls a runtime library to overcome this problem, even in cases where the dividend is 4-byte, the divisor is 2-byte, and the result is 2-byte.

-fuse_MUL -fUM

Modify generated code

Function: Changes generated code for multiplication operations.

Supplement: When 16 bits × 16 bits is to be stored in 32 bits, either of the multiplier or multiplicand needs to be cast in 32 bits in order to obtain the result consisting of the 16 high-order bits.
By specifying this option, it is possible to obtain a full 32-bit result without the need for a cast.

-R8C

Modify generated code

Function: Generates code appropriate for the R8C family MCU.

Supplement: When this option is specified, the -fnear_ROM(f-NROM) option is always enabled.
When this option is specified, the keywords far and _far are ignored.
When this option is selected, the libraries linked become as follows:

a) If a project is built in the integrated environment (High-performance Embedded Workshop), the library generator automatically generates a library and optlnk links that library.
b) If nc30.exe is invoked at the command prompt and executed until after a link process, nc30.exe automatically links the following libraries.
   - When the -fsizet_16 or -fptrdifft_16 option is used simultaneously r8cs16.lib
   - Otherwise r8clib.lib
c) If optlnk is invoked at the command prompt separately, link the library generated by the library generator.

Notes:
(1) This option cannot be used in combination with the options listed below.
- ffar_RAM(- fFRAM)
- fno_carry(- fNC)
- fchange_bank_always(- fCBA)
(2) Add this option for all programs you link.

-R8CE

Modify generated code

Function: Generates code appropriate for the R8C family MCU (ROM 64 Kbytes or larger).

Notes:

(1) This option cannot be used in combination with the options listed below.
-ffar_RAM(-fFRAM)
-fno_carry(-fNC)
-fchange_bank_always(-fCBA)

(2) Use this option when the ROM area exceeds the 64-Kbyte boundary.

(3) When this option is selected, the libraries linked become as follows:

a) If a project is built in the integrated environment (High-performance Embedded Workshop), the library generator automatically generates a library and optlnk links that library.

b) If nc30.exe is invoked at the command prompt and executed until after a link process, nc30.exe automatically links the following libraries.

- When the -fsizet_16 or -fptrdifft_16 option is used simultaneously r8ces16.lib
- Otherwise r8celib.lib

c) If optlnk is invoked at the command prompt separately, link the library generated by the library generator.

(4) Add this option for all programs you link.

-fSB_auto -fSBA

Modify generated code

Function: Switches the SB register from one to another before generating SB relative addressing, one function at a time.

Supplement: The number of times external variables are referenced is analyzed to generate optimum SB relative addressing, one function at a time.

```
int sym;
int a;
int data;
   :
int b;
   :
int func(void){
      a = x;
      sym = xx;         ----->  The _sym address is made the base
      sym = a*b;                point for SB relative.
      if( sym != 0)
              sym = sub( );
      return sym;
}

int sub( void)
{
       data1 = sym1;     ----->  The _data1 address is made the base
       data2 = data1/2;          point for SB relative.
       data1 = sub(data2);
           :
```

(1) The address of the symbol that is made the base point for SB relative is stored in the SB register.
(2) Code is generated that saves and restores the SB register at the entry and exit to and from the function.
(3) Effective for only external variables.
(4) This option cannot be used in combination with -OR, -OS, -OR_MAX(-ORM), and -OS_MAX(-OSM).
(5) Behavior of a program that has linked in it the object files using this option and those using the facilities given below is not guaranteed.
- #pragma SBDATA
- Compiler option -fauto_over_255(-fAO2)

(6) The -goptimize option cannot be specified at the same time.
(7) The -finfo option is enabled.
(8) Do not use this option for the compiler sources that load the SBDATA declaration headers generated by utl30.

A.2.7 Library Specifying Options

-l *library file name*

Function: Specifies a library file name that is used by optlnk when lining files. The file extension can be omitted.

Syntax: nc30△-l filename△<C/C++ source file name>

Notes:
(1) In file specification, the extension can be omitted. If the extension of a file is omitted, the file is processed assuming that it has the extension ".lib."
(2) To specify a file extension, be sure to specify ".lib."
(3) Files are searched in the current folder and the folder specified by the environment variable HLINK_DIR in that order.
(4) NC30 links, by default, a library "nc30lib.lib" present in the directory that is specified by the environment variable LIB30. (If the compile option "-R8C," "-R8CE," "-fsizet_16," or "-fptrdifft_16" is specified, refer to the description of each option.)
(5) If a library is specified by this option, the library (4) linked by default by NC30 is assigned the lowest priority.

A.2.8 Warning Options

-Wall

Warning option

Function: Displays all detectable warnings.

Supplement:
(1) The warnings displayed here do not include those that may be generated when "Wlarge_to_small(-WLTS)," "Wno_used_argument(-WNUA)," and "Wno_used_static_function(-WNUSF)" are used.
(2) The warnings displayed here are equivalent to "Wnon_prototype(-WNP)," "Wunknown_pragma(-WUP)," "Wnesting_comment(-WNC)," and "Wuninitialize_variable(-WUV)."
(3) Warnings are displayed in the following cases too:
- When the assignment operator "=" is used in if statements, for statements, or comparison statements of the && or | | operator.
- When the assignment operator "=" is erroneously written as "= =."
- When any function in old format is defined.

Notes: These warnings are detected within the scope that the compiler assumes on its judgment that description is erroneous. Therefore, not all errors can be warned.

-Wccom_max_warnings=*warning count* -WCMW=*warning count*

Warning option

Function: Allows you to specify an upper limit for the number of warnings output by the compiler.

Supplement: By default, there is no upper limit to warning outputs.
Use this option to adjust the screen as it scrolls for many warnings that are output.

Notes: For the upper-limit count of warning outputs, specify a number equal to or greater than 0. Specification of this count cannot be omitted. When a count of 0 is specified, warning outputs are completely suppressed.
This facility is effective only when the input file is compiled as a C program.

-Wlarge_to_small -WLTS

Warning option

Function: Outputs a warning about implicit assignments from large size to small size.

Supplement: A warning may be output for boundary values of negative numbers of any type even when they fit in the type. This is because negative numbers are considered under language conventions to be an integer combined with the unary operator (-).
For example, the value -32768 fits in the signed int type, but when broken into "-" and "32768," the number 32768 does not fit in the signed int type and, consequently, becomes the "signed long type." Therefore, the immediate "-32768" is the signed long type.
For this reason, any statement like "int i = -32768;" gives rise to a warning.

Notes: Because this option outputs a large amount of warnings, warning output is suppressed for the type conversions listed below.

- Assignment from char type variables to char type variables
- Assignment of immediates to char type variables
- Assignment of immediates to float type variables

-Wnesting_comment -WNC

Warning option

Function: Generates a warning when comments include "/*."

Supplement: By using this option, it is possible to detect nesting of comments.

-Wno_stop -WNS

Warning option

Function: Prevents the compiler from stopping when an error occurs.

Supplement: The compiler compiles the program one function at a time. If an error occurs when compiling, the compiler by default does not compile the next function.
Also, an error may cause another error to occur, giving rise to multiple errors. In such a case, the compiler stops compiling.
When this option is specified, the compiler continues compiling as far as possible.

Notes: A system error may occur due to erroneous description in the program. In such a case, the compiler stops compiling even when this option is specified.
If, when compiled as a C++ program, a total of 100 errors have been output, the compiler stops compiling, regardless of whether this option is specified.

Precautions concerning the compiler option -Wlarge_to_small(-WLTS)

When you use the compiler option -Wlarge_to_small(-WLTS), pay attention to the following.

(1) When compiled as a C++ program, a warning is output only when the right side is a constant.

(2) When compiled as a C program, a warning is output when the right side consists only of a variable.

-Wno_used_argument -WNUA

Warning option

Function: When a function that has parameters is defined, this option outputs a warning for unused parameters in it.

-Wno_used_function -WNUF

Warning option

Function: Displays unused global functions when linking.

Notes: If the -msg_unused option is specified when linking, this option is unnecessary.
If the linker options -msg_unused and -message are used, this option is unnecessary.

-Wno_used_static_function -WNUSF

Warning option

Function: Displays the names of static functions that do not require code generation. This message is output when one of the following conditions applies.

- The static function is not referenced from anywhere in the file.
- The static function is turned into inline by use of the "-Ostatic_to_inline(-OSTI)" option.

Notes: (1) If any function name is written in the initializer of an array as shown below, the compiler will process the function assuming that it will be referenced, even though it may not actually be referenced during program execution.

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

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

-Wno_warning_stdlib -WNWS

Warning option

Function: This option, when selected simultaneously with "-Wnon_prototype" or "-Wall," suppresses warnings for "standard libraries that do not have prototype declarations."

Supplement: If, when compiled as a C++ program, no prototype declarations are found, the compiler outputs a message, regardless of whether this option is specified.

-Wnon_prototype -WNP

Warning option

Function: Outputs a warning when a function is used that has its prototype not declared prior to the call or there is no prototype declaration for the function in the source.

Supplement: Declaring prototype for a function permits arguments to the function to be passed via a register. Increased speed and reduced code size can be expected by passing arguments via a register. Also, a prototype declaration causes the compiler to inspect parameters of the function. Increased program reliability can be expected from this.
Therefore, Renesas recommends using this option whenever possible.
If, when compiled as a C++ program, no prototype declarations are found, the compiler outputs a message, regardless of whether this option is specified.

-Wstop_at_link -WSAL

Warning option

Function: Changes all information and warning messages to the error level when linking.
The compiler aborts a link process when an error message is output.

-Wstop_at_warning -WSAW

Warning option

Function: Stops compiling when a warning occurs during compilation.

-Wundefined_macro -WUM

Warning option

Function: Outputs a warning when an undefined macro is used in #if.

-Wuninitialize_variable -WUV

Warning option

Function: Outputs a warning for uninitialized auto variables.
This option is effective even when "-Wall" is specified.

Supplement: If an auto variable is initialized in conditional jump by, for example, a if statement or a for statement in the user application, the compiler assumes that the variable has not been initialized.
Therefore, when this option is used, the compiler outputs a warning for it.

-Wunknown_pragma -WUP

Warning option

Function: Outputs a warning when an unsupported #pragma is used.

Supplement: By default, no warnings are output even when an unsupported, unknown "#pragma" is used.
When using only the NC-series compilers, this option helps to find misspellings in "#pragma."

Notes: When you are using only the NC-series compilers, Renesas recommends that this option be always used when compiling.

A.2.9 Assemble and Link Options

-as30 "*option*"

Assemble/link option

Function: Selects as30 assemble command options
To select two or more options, enclose them in double quotes.

Syntax: nc30△-as30△"option1Δoption2"△<C source file>

Notes: Do not specify the as30 options "-.", "-C", "-M", "-O", "-P", "-T", "-V," or "-X".

-lnkcmd=command file name

Assemble/link option

Function: Specifies a command file for optlnk. It is passed as -subcommand option to optlnk.

Syntax: nc30△-lnkcmd=command filename>△<C/C++ source file>
For the command file format, refer to a section in which the -subcommand option of optlnk is described.

A.3 Notes on Startup Options

A.3.1 Notes on Description of Startup Options

The startup options of nc30 are discriminated according to whether they are written in uppercase or lowercase letters. The functionality of an option is nullified when it is specified in the wrong case.

A.3.2 Priority of Options

If the following startup options of nc30 are specified at the same time,

- "-c": Creates object files (extension ".obj") and finishes processing.
- "-S": Creates assembler source files (extension ".a30") and finishes processing.

then the -S option takes precedence and only the assembler source files will be generated.

Appendix B Extended Functions Reference

To facilitate its use in systems using the M16C Series, R8C Family, NC30 has a number of additional (extended) functions.

This appendix B describes how to use these extended functions, excluding those related to language specifications, which are only described in outline.

This compiler, in addition to the keywords in standard language specifications, handles the following as extended keywords.

_asm, _far, _inline, _near, asm (standard keyword in C++), far, inline (standard keyword in C++), near, _Bool (C only), restrict (C only), and _ext4mptr (C only)

Table B.1 Extended Functions (1/2)

Extended feature	Description
near/far qualifiers	Specifies the addressing mode to access data. near..... Access to an area within 64K bytes (0H to 0FFFFH) far........ Access to an area beyond 64K bytes (all memory areas). • All functions take on far attributes.

asm function	(1) Assembly language can be directly included in C/C++ programs. It can also be included outside functions.
	Example: asm(" MOV.W #0, R0");
	(2) You can specify variable names (within functions only).
	Example1:
	asm(" MOV.W R0, $$[FB]",f);
	Example2:
	asm(" MOV.W R0, $$",s);
	Example3:
	asm(" MOV.W R0, $@",f);
	(3) You can include dummy asm functions as a means of partially suppressing optimization (within functions only).
	Example: asm( );

Japanese characters	(1) Permits you to use Japanese characters in character strings. Example: L" 漢字 " (2) Permits you to use Japanese characters for character constants. Example: L' 漢 ' (3) Permits you to write Japanese characters in comments. Example: / * 漢字 */ • Shift-JIS and EUC code are supported ,but can't use the half size character of Japanese-KATA-KANA

Table B.2 Extended Functions (2/2)

Extended feature	Description

Default argument declaration for function	Default value can be defined for the argument of a function.
	Example1:
	extern int func(int=1, char=0);
	Example2:
	extern int func(int=a, char=0);
	• When writing a variable as a default value, be sure to declare the variable used as a default value before declaring the function.
	• Write default values sequentially beginning immediately after the argument.
	• This feature is an extended provided for compiling in C mode. When in C++ mode, C++ language specification applies.

Inline storage class	Functions can be inline developed by using the inline storage class specifier.inline. Example: inline func(int i); • This feature is an extended provided for compiling in C mode. When in C++ mode, C++ language specification applies.
#pragma Extended functions	Extended features for making the most of M16C series or R8C family specifications can be used.
macro assebler function	Part of assembly language can be written as a function.
Binary integer constant	Binary numbers can be written using integer constants. Write a string of numerals 0 and 1 immediately following 0B or 0b. Example: 0b01011100
Long long type	It is possible to handle long long type. To write an integer constant of long long type, add an LL or ll suffix after the constant value. Example: 123456789012LL
_Bool type	It is possible to handle _Bool type. _Bool type represents 0 or 1.
C++ comments	A C++ comment (//) can be written in a C program.

B.1 Near and far Modifiers

For the M16C Series, R8C Family microcomputers, the addressing modes used for referencing and locating data vary around the boundary address 0FFFFH. NC30 allows you to control addressing mode switching by near and far qualifiers.

B.1.1 Overview of near and far Modifiers

The near and far qualifiers select an addressing mode used for variables or functions.

(1) near modifier................................ Area of 000000H to 00FFFFH
(2) far modifier....................................Area of 000000H to 0FFFFFH

The near and far modifiers are added to a type specifier when declaring a variable or function.If you do not specify the near or far modifiers when declaring variables and functions, NC30 interprets their attributes as follows:

(1) Location of variables ....near attribute
(2) Location of const-qualified variables.............far attribute
(3) Location of functions..far attribute

Furthermore, NC30 allows you to modify these default attributes by using the startup options of compile driver nc30.

B.1.2 Format of Variable Declaration

The near and far modifiers are included in declarations using the same syntactical format as the const and volatile type modifiers. Figure B.1 is a format of variable declaration.

```
type specifier△near or far△variable;
```

Figure B.1 Format of Variable added near / far modifier

Figure B.2 is an example of variable declaration. Figure B.3 is a memory map for that variable

```
int near    in_data;
int far     if_data;

void        func(void)
{
            (remainder omitted)
                :
```

Figure B.2 Example of Variable Declaration

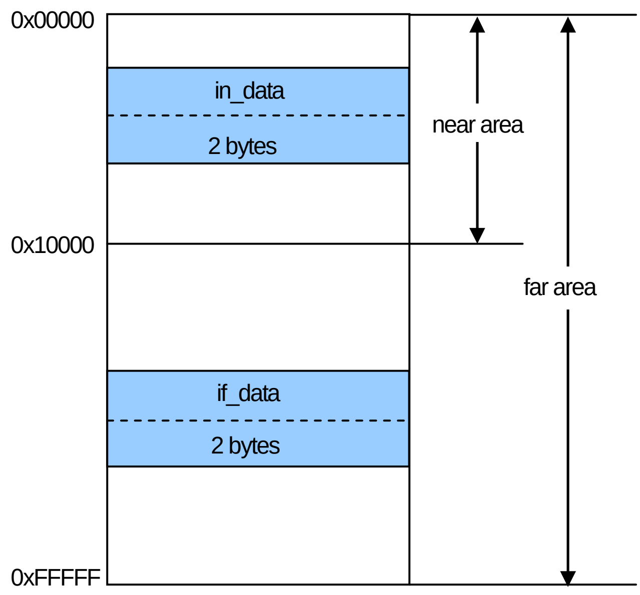


Figure B.3 Memory Location of Variable

B.1.3 Format of Pointer type Variable

Pointer-type variables by default are the near-type (2-byte) variable. A declaration example of pointer-type variables is shown in Figure B.4.

```
Example:

        int         * ptr;
```

Figure B.4 Example of Declaring a Pointer Type Variable (1)

Because the variables are located near and take on the pointer variable type near, the description in Figure B.4 is interpreted as in Figure B.5.

```
Example:

        int         near* near  ptr;
```

Figure B.5 Example of Declaring a Pointer Type Variable (2)

The variable ptr is a 2-byte variable that indicates the int-type variable located in the near area. The ptr itself is located in the near area.Memory mapping for the above example is shown in Figure B.6.

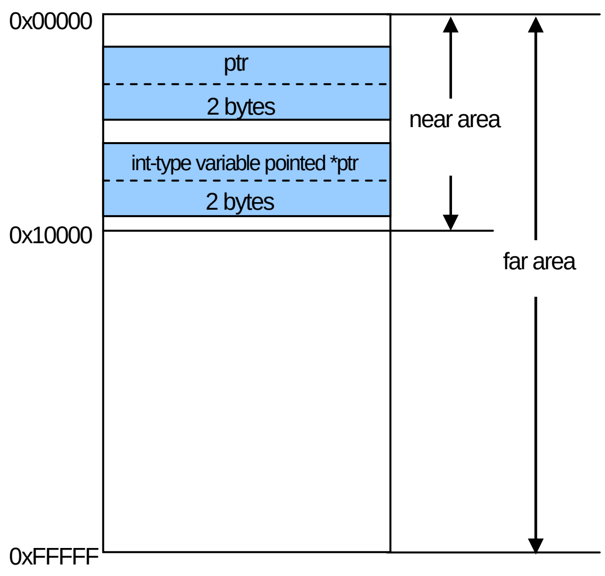


Figure B.6 Memory Location of Pointer type Variable

When "near and far" is explicitly specified, determine the size of the address at which to store the "variable and function" that is written on the right side. A declaration of pointer-type variables that handle addresses is shown in Figure B.7.

```
Example1:
        int     far *     ptr1;
Example2:
        int     * far     ptr2;
```

Figure B.7 Example of Declaring a Pointer Type Variable (3)

As explained earlier, unless "near and far" is specified, the compiler handles the variable location as "near" and the variable type as "far." Therefore, Examples 1 and 2 respectively are interpreted as shown in Figure B.8.

```
Example1:
        int     far * near   ptr1;
Example2:
        int     near * far   ptr2;
```

Figure B.8 Example of Declaring a Pointer Type Variable (4)

In Example 1, the variable ptr1 is a 4-byte variable that indicates the int-type variable located in the far area. The variable itself is located in the near area.
In Example 2, the variable ptr2 is a 4-byte variable that indicates the int-type variable located in the far area. The variable itself is located in the far area.Memory mappings for Examples 1 and 2 are shown in Figure B.9.

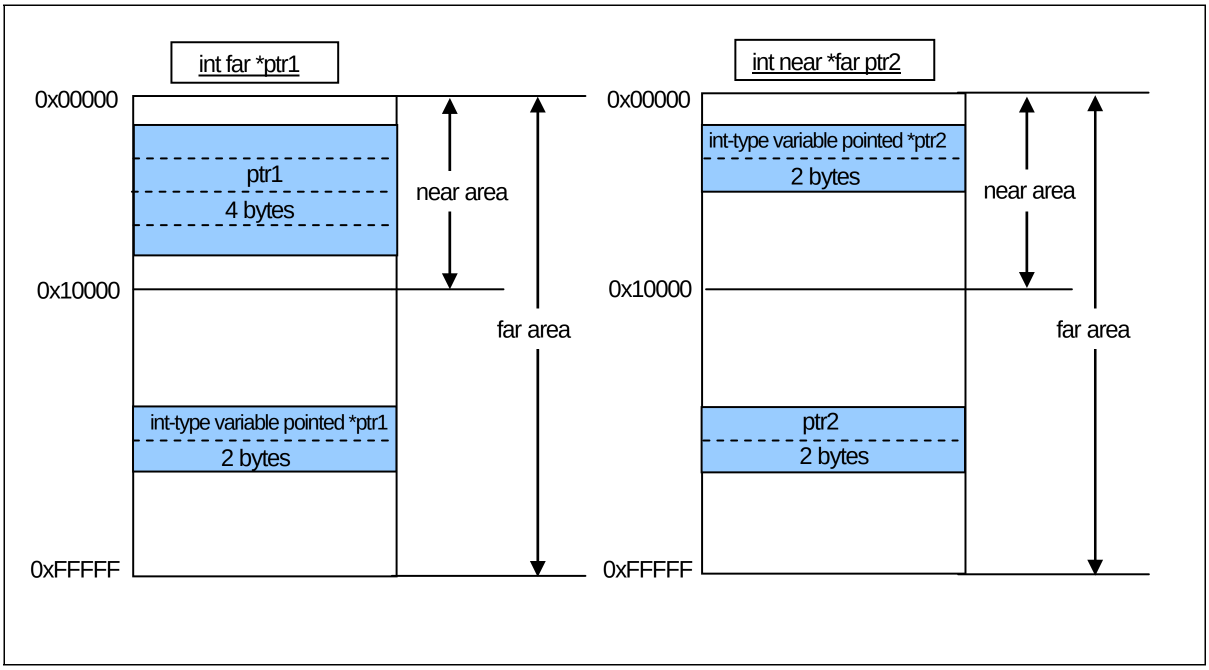


Figure B.9 Memory Location of Pointer type Variable

B.1.4 Format of Function Declaration

For C, specifying the near attribute in a function declaration causes a warning message to be output, with the near declaration ignored.
When compiled as C++, near/far location attributes for functions result in an error.
When compiled as C++, overloaded definitions of functions by near/far attributes are handled in the same way as for the const attribute.

```
void func(int near * np)   { ... }
void func(int far   * fp)  { ... }     // The near/far attribute pointers can be overloaded.
void func(int near n)      { ... }
void func(int far   f)     { ... }     // near/far-attribute locations cannot be overloaded. An error is assumed.
```

B.1.5 near and far Control by nc30 Command Line Options

If near/far attributes are not specified, NC30 handles functions as having the attribute "far" and variables as having the attribute "near." NC30 has options available that change the default near/far attributes of variables (data) or pointers. (See Table B.3.)

Table B.3 Command Line Options

Command Line Options	Function
-fnear_ROM(-fNROM)	Assumes near as the default attribute of ROM data
-ffar_RAM(-fFRAM)	Assumes far as the default attribute of RAM data.
-fconst_not_ROM(-fCNR)	Does not handle const-qualified types as ROM data.
-ffar_pointer(-fFP)	Assumes the attribute "far" for the default attribute of pointer type.
-R8C	Changes the default attribute of ROM data to "near." The far/_far attributes are ignored.

B.1.6 Function of Type conversion from near to far

The program in Figure B.10 performs a type conversion from near to far.

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

void        main(void)
{
            f_ptr = n_ptr;          /* assigns the near pointer to the far pointer */
                :
            (abbreviated)
                :
            func ( n_ptr );         /* For a function which has had its prototype declared as having a
                                       far pointer for parameter */
                                    /* specifies near pointer parameter at the function call */
}
```

Figure B.10 Type conversion from near to far

To convert types to far, set zeros in the 2 upper bytes.

B.1.7 Checking Function for Assigning far Pointer to near Pointer

When compiled as a C program, the compiler outputs a warning message "assign far pointer to near pointer, bank value ignored" regarding a program fragment written as in Figure B.11, indicating that the upper address byte (bank value) will be lost.

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

void        main(void)
{
            n_ptr = f_ptr;                  /* Assigns a far pointer to a near pointer */
                :
            (abbreviated)
                :
            func f_ptr;                     /* For a function which has had its prototype declared as having a
                                               near pointer for parameter */
                                            /* far pointer implicitly cast as near type */
            n_ptr = (near *)f_ptr;          /* far pointer explicitly cast as near type */
}
```

Figure B.11 Type conversion from far to near

The warning message "far pointer (implicitly) casted by near pointer" is also output when a far pointer is explicitly cast as a near pointer, then assigned to a near pointer.
When compiled as a C++ program, an assignment from the far pointer to the near pointer results in an error.
If it is evident that the far pointer to be substituted points to a near area, cast the far pointer to the near pointer by a cast notation or const_cast operator in order to avoid an error.
Note that dynamic_cast, static_cast, and reinterpret_cast cannot change near/far types.

B.1.8 Class Declarations by near/far

This compiler, when compiling C++, permits near/far qualifications in a class declaration.
The near/far-qualified class "this" has the same attribute as near/far qualifiers in the class declaration, irrespective of the default attribute of RAM data pointers.

```
class _far foo {
  public:
     .....
     .....
};
```

Figure B.12 Example of a Class Declaration by near/far

The near/far attribute of "this" pointer is the same as the default attribute of RAM data pointers.
Because of this specification, the following limitations apply.

- Since conversion from a far pointer to a near pointer will be produced if the default attribute of the RAM data pointer is 'near', calling a non-static member function from a variable with the far attribute may lead to an error.
- To link the object files where the default attributes of RAM data pointers are "near," the heap area needs to be located in a near area.
- Do not link the object files where the same class is declared and the default attributes of RAM data pointers are different.

Note that near/far qualifications in a class declaration make it possible to separate the near/far attributes of "this" pointer from the default attributes of RAM data pointers.

B.1.9 Template Functions and near/far Declarations

The template functions that can be overloaded with near/far attributes take on the attribute "near" and the attribute "far."
If a template parameter is qualified as near, it cannot be instantiated by a far-qualified template argument.
If a template parameter has the far attribute and a template argument has the near attribute, then the template argument is extended to the far attribute when it is instantiated.

B.1.10 Function for Specifying near and far in Multiple Declarations

When multiple declarations for one and the same variable are compiled as a C program as in Figure B.13, information about the type of these variables is interpreted as combined type.

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

void         func(void)
{
             (remainder omitted)
                  :


This Declaration is interpreted as the following:

extern int   far idata = 10;

void         func(void)
{
             (remainder omitted)
                  :
```

Figure B.13   Integrated Function of Variable Declaration

As shown in this example, if there are many declarations, the type can be declared by specifying "near or far" in one of those declarations. However, an error occurs if there is any contention between near and far specifications in two or more of those declarations.
You can ensure consistency among source files by declaring "near or far" using a com¬mon header file.

common header file
common.h

```
extern int   far data;
```

C source file
a.c

```
#include "common.h"

void         main(void)
{
             data = 1;
}
```

C source file
b.c

```
#include "common.h"

int          data = 10;
```

Figure B.14   Example of Common header file Declaration

When compiled as a C++ program, a fragment to determine near/far in multiple declarations that is accepted when compiled as a C program may cause an error.

```
extern int far fi;
int fi;         // For C, the type of fi is interpreted as int far.
                // For C++, if the RAM data location attribute is far, the type of fi is interpreted as int far;
                // if the RAM data location attribute is near, an error is assumed.

extern int near ni;
int ni;         // For C, the type of fi is interpreted as int near.
                // For C++, if the RAM data location attribute is far, an error is assumed;
                // if the RAM data location attribute is near, the type of ni is interpreted as int near.

extern int far * fpi;
int * fpi;      // For C, the type of fpi is interpreted as int far.*
                // For C++, if the RAM data pointer attribute is far, the type of fpi is interpreted as int far*;
                // if the RAM data pointer attribute is near, an error is assumed.

extern int near * npi;
int * npi;      // For C, the type of npi is interpreted as int near*.
                // For C++, if the RAM data location attribute is far, an error is assumed;
                // if the RAM data pointer attribute is near, the type of npi is interpreted as int near*.
```

B.1.11 Notes on near and far Attributes

a. Notes on near and far Modifier Syntax

Syntactically, the near and far modifiers are identical to the const modifier.The following code therefore results in an error.

```
int     i, far    j;      ← This is not permitted
   ▼
int     i;
int     far j;
```

Figure B.15 Example of Variable Declaration

- For C++, the location attribute of struct, union, or class member variables cannot be near/far-qualified.
- For C++, the struct, union, or class member variables that are specified as mutable, even when the class objects are const-specified, will have their const specification removed.
- For C++, if a variable that has reference type to near-qualified type is initialized with a far-attribute variable, an error results.
- For C++, locations indicated by pointers to struct, union, or class members cannot be near/far-qualified. Such specification results in an error.

B.2 asm Function

NC30 allows you to include assembly language routines (asm functions)[1] in your C/C++ language source programs.

B.2.1 Overview of asm Function

The asm function is used for including assembly language code in a C/C++ language source program. As shown in Figure B.16, the format of the asm function is asm(" ");, where an assembly language instruction that conforms to the AS30 language specifications is included between the double quote marks.

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

void      func(void)
{
              :
          (abbreviated)
              :
          ta0_int = 0x07;                 ← Permits timer A0 interrupt

          asm("       FSET I");           ← Set interrupt enable flag
}
```

Figure B.16 Example of Description of asm Function (1)

Compiler optimization based on the positional relationship of the statements can be partially suppressed using the code shown in Figure B.17.

```
asm( );
```

Figure B.17 Example of Coding asm Function(2)

The asm function used in NC30 not only allows you to include assembly language code but also has the following extended functions:

- The FB offset values for variables of storage class "auto" in a C/C++ program can be specified by a C/C++ language variable name.
- The register names for variables of storage class "register" in a C/C++ program can be specified by a C/C++ language variable name.
- The symbol names for variables of storage classes "extern" and "static" in a C/C++ program can be specified by a C/C++ language variable name.

Described below are the precautions to be taken when using asm functions.

- The compiler does not check the registers that are altered in asm functions.
- To alter registers, write push and pop instructions using asm functions to save and restore the registers.
- The symbols that begin with ‘$’ or ‘_’ are the reserved symbols for the compiler. Behavior of a program where a definition of symbols beginning with ‘$’ or ‘_’ is written in an asm function cannot be guaranteed.
- Do not write the directive ".section" in asm functions. To change section names, always use #pragma SECTION outside the asm functions.

[1] For the purpose of expression in this user's manual, the subroutines written in the assembly language are referred to as assembler functions. Those written with asm() in a C language program are referred to as asm functions or inline assemble description.

B.2.2 Specifying FB Offset Value of auto Variable

The variables (parameters included) of storage classes "auto" and "register" are referenced and located by an offset value relative to the frame base register (FB). (It is possible that they will be assigned to registers by optimization, etc.)
The auto variables which are mapped to the stack can be used in the asm function by writing the program as shown in Figure B.18 below.

```
asm( "    opecode    opeland , $$ [ FB ] " , variable name );
```

Figure B.18 Descroption Format for Specifying FB Offset

Only two variable name can be specified by using this description format. The following types are supported for variable names:

- Variable name
- Array name [integer]
- Struct name, member name (not including bit-field members)

```
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, $$[FB]", s.i);
                        (Remainder omitted)
                           :
            asm("       MOV.W       $$[FB], $$[FB]", s.i, a[2]);
}
```

Figure B.19 Description example for specifying

Figure B.20 shows an example for referencing an auto variable and its compile result.

```
C source file:

void        func(void)
{
            int idata = 1;          ← auto variable(FB offset value =-2)

            asm("       MOV.W       $$[FB], R0", idata);
            asm("       CMP.W       #00001H ,R0");
            (remainder omitted)
                    :
}


Assember lsource file (compile result):

;## # FUNCTION func
;## # FRAME AUTO ( idata) size 2, offset -2
                :
            (abbreviated)
;## # C_SRC : asm("    MOV.W       $$[FB], R0", idata);
;#### ASM START
            MOV.W       -2[FB], R0  ← Transfer FB offset value-2 to R0 register
            ._line 5
;## # C_SRC : asm(" CMP.W #00001H,R0");
            CMP.W       #00001H ,R0
;#### ASM END
            (remainder omitted)
                    :
```

Figure B.20　Example for Referencing an auto Variables

You can also use the format show in Figure B.21 so that auto variables in an asm function use a 1-bit field. (Can not operate bit-fields greater than 2-bits.)

```
asm( "      opecode       $b[ FB ]" , bit field name);
```

Figure B.21　Format for Specifying FB Offset Bit Position.

You can only specify one variable name using this format. Figure B.22 is an example.

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

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

Figure B.22　Example for Specifying FB Offset Position

Figure B.23 shows examples of referencing auto area bit fields and the result of compiling.

C source file:

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

Assembler source file(compile result):

```
;## # 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'
            ._line 3
            .glb        _func
_func:
            enter       #02H
            ._line 10
;#### ASM START
            bset        1,-2[FB] ; s
;#### ASM END
            ._line 11
            exitd
```

Figure B.23 Example of Referencing auto Area Bit Field

To reference the bit-field in an auto area, check to see that it is located within the range referenceable by bit processing (the range within 32 bytes centering on FB register value).

B.2.3 Specifying Register Name of register Variable

It is possible that the variables (parameters included) of storage classes "auto" and "register" will be assigned to registers by the compiler.
The variables mapped to registers can be used in the asm function by writing the program as shown in Figure B.24 below.[1]

```
asm( "        opecode    opeland, $$ " , variable name);
```

Figure B.24 Description Format for Register Variables

You can only specify two variable name using this format.Figure B.25 shows examples of referencing register variables and the results of compiling.

C Source file:

```
void        func(void)
{
            register int  i=1;

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

Assembler source file (compile result):

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

Figure B.25 An Example for Referencing a Register Variable and its Compile Result

In NC30, register variables used within functions are managed dynamically. At anyone position, the register used for a register variable is not necessarily always the same one. Therefore, if a register is specified directly in an asm function, it may after compiling operate differently. We therefore strongly suggest using this function to check the register variables.

[1] *1 If the variables need to be forcibly mapped to registers using the register qualifier, specify the option -fenable_register (-fER) when compiling.

B.2.4 Specifying Symbol Name of extern and static Variable

The variables of storage classes "extern" and "static" are referenced as symbols.
You can use the format shown in Figure B.26 to use extern and static variables in asm functions.

```
asm( "     opcode     opeland   $$ " , variable name );
```

Figure B.26 Description Format for Specifying Symbol Name

Up to two variables can be specified in this command form. Following are supported as variable names:

- Variable name
- Array names [constants]
- Struct name, member name (not including bit-field members)

```
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);
            (remainder omitted)
                 :
}
```

Figure B.27 Description example for specifying

See Figure B.28 for examples of referencing extern and static variables.

```
C source file:
extern int    ext_val;      ←extern variable

void          func(void)
{
              static int    s_val;

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


Assembler source file(compile result):
_func:
              ._line 7
;## # C_SRC : asm("    mov.w        #01H,$$",ext_val);
;#### ASM START
              mov.w      #01H,_ext_val                   ← Transfer to the extern variable "ext_val"
              ._line 8
;## # C_SRC : asm(" mov.w           #01H,$$",s_val);
              mov.w      #01H,___S0_s_val               ← Transfer to the intra-function static variable "s_val"
;#### ASM END
              ._line 9
;## # C_SRC : }
              rts
E1:
              .glb        _ext_val
              .section    bss_NE,DATA
___S0_s_val: ;### C's name is s_val
              .blkb 2
              .END
```

Figure B.28 Example of Referencing extern and static Variables

You can use the format shown in Figure B.29 to use 1-bit bit fields of extern and static variables in asm functions.(Can not operate bit-fields greater than 2-bits.)

```
asm( "      opecode      $b", bit field name );
```

Figure B.29 Format for Specifying Symbol Names

You can specify one variable name using this format. See Figure B.30 for an example.

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

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

Figure B.30 Example of Specifying Symbol Bit Position

Figure B.31 shows the results of compiling the C source file shown in Figure B.30.

```
;## # 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        ← Reference to bitfield bit0 of structure s
;#### ASM END
        ._line 12
;## # C_SRC : }
        rts
E1:
        .align
        .section    bss_NO,DATA
        .glb        _s
_s:
        .blkb 1
        .END
```

Figure B.31 Example of Referencing Bit Field of Symbol

To reference the bit-fields of extern or static variables, check to see that they are located in the range referenceable by instructions for absolute bit instruction addressing (the range from 0000H to 1FFFH).

B.2.5 Specification Not Dependent on Storage Class

Variables can be used in asm functions independently of their storage classes (auto, register[1] , extern, or static).
Using the command syntax shown in Figure B.32, it is possible to use variables in asm functions[2].

```
asm("      opecode   opeland, $@", variable name );
```

Figure B.32 Description Format Not Dependent on Variable's Storage Class

You can specify two variables name using this format. Figure B.33 shows examples of referencing register variables and the results of compiling.

```
C source file:
extern int   e_val;

void         func(void)
{
                int          f_val; .     ← auto variable
             register int f_val;          ←register variable
             static int   s_val;          ←static variable

             asm("        mov.w       #1, $@", e_val);       ← Reference to external variable
             asm("        mov.w       #2, $@", f_val);       ← Reference to auto variable
             asm("        mov.w       #3, $@", r_val);       ← Reference to register variable
             asm("        mov.w       #4, $@", s_val);       ← Reference to static variable
             asm("        mov.w       $@, $@", f_val,r_val);
}

Assembler source file(compile result):
             .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                      ← Reference to external variable
             ._line 10
;## # C_SRC : asm("        mov.w       #2, $@", f_val);
             mov.w        #2, -2[FB]                         ← Reference to auto variable
             ._line 11
;## # C_SRC : asm("        mov.w       #3, $@", r_val);
             mov.w        #3, R1                             ← Reference to register variable
             ._line 12
;## # C_SRC : asm("        mov.w       #4, $@", s_val);
             mov.w        #4, ___S0_s_val:16                 ← Reference to static variable
             ._line 13
;## # C_SRC : asm("        mov.w       $@, $@", f_val,r_val);
             mov.w        -2[FB], R1
;#### ASM END
```

Figure B.33 Example for Referencing Variables of Each Storage Class

[1] It does not restrict being assigned to a register, even if it specifies a register qualified.
[2] Whether it is arranged at which storage class should actually compile, and please check it.

B.2.6 Method for Suppressing Optimization Partially

In Figure B.34, the dummy asm function is used to selectively suppress a part of optimization.

```
#pragma  ADDRESS  port  02H
struct port{
          char      bit0:1;
          char      bit1:1;
          char      bit2:1;
          char      bit3:1;
          char      bit4:1;
          char      bit5:1;
          char      bit6:1;
          char      bit7:1;
}port;

void      func(void)
{
          port.bit0 = 0x01;
          port.bit1 = 0x01;
}

          port.bit0 = 0x01;
          asm();     /*dummy*/
          port.bit1 = 0x01;
```

Optimization results in any steps to set the two port bits separately being combined as one step.

Optimization ▶

```
or.b #03H,_port
```

Optimization ▶

```
bset 00H,_port
bset 01H,_port
```

Optimization is suppressed.

Figure B.34 Example of Suppressing Optimization by Dummy asm

B.2.7 Notes on the asm Function

a. Extended Features Concerning asm functions

When using the asm function for the following processing, be sure to use the format shown in the coding examples.

(1) For stack variables

Do not use an offset value from the frame base register (FB) to specify stack variables (including parameters). Write the specification for stack variables in the command form shown in Figure B.35.

```
asm("    MOV.W     #01H,$$[FB]", i);     ← Format for referencing auto variables
asm("    BSET      $b [FB]", s.bit0);    ← Format for checking auto bit fields
```

Figure B.35 Example Coding of asm Function (1)

(2) Specification of register storage class

You can specify the register storage class in NC30. When register class variables are compiled with option -fenable_register (-fER), use the format shown in Figure B.36 for register variables in asm functions.

```
asm("    MOV.W    #0,$$", i);        ← Format for checking register variables
```

Figure B.36 Example Coding of asm Function (2)

Also, if the option -O[1–5], -OR, -OS, -OR_MAX(-ORM), or -OS_MAX(-OSM) is specified, the arguments to be passed via register may be handled as register variables without transferring them to the auto area for the sake of increased code efficiency.

In this case, when parameters are specified in an asm function, the assembly language is output using the register names instead of the variable's FB offset.

(3) When referencing arguments in the asm function

The compiler analyzes program flow in the interval in which variables (including arguments and auto variables) are effective, as it processes the program. For this reason, if arguments or auto variables are referenced directly in the asm function, management of such effective interval is destroyed and the compiler cannot output codes correctly.

Therefore, to reference arguments or auto variables in the asm function you are writing, always be sure to use the "$$, $b, $@" features of the asm function.

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

Figure B.37 Example cannot be referred to correctly

In the above case, because the compiler determines that "i" and "j" are not used within the function func, it does not output codes necessary to construct the frame in which to reference the arguments. For this reason, the arguments cannot be referenced correctly.

(4) About branching within the asm function

The compiler analyzes program flow in the intervals in which registers and variables respectively are effective, as it processes the program. Do not write statements for branching (including conditional branching) in the asm function that may affect the program flow.

b. About Register

- Do not alter registers in asm functions. If it is necessary to alter, use the push/pop instructions to save/restore the registers.
- NC30 is premised on condition that the SB register is used in fixed mode after being initialized by the startup program. If you modified the SB register, write a statement to restore it at the end of consecutive asm functions as shown in Figure B.38.

```
asm("     .SB        0);
asm("     LDC        #0H, SB");          ← SB changed
asm("     MOV.W      R0, _port[SB]");
               :
          (abbreviated)
               :
asm("     .SB        __SB__);
asm("     LDC        #__SB__,SB");       ←SB returned to original state
```

Figure B.38 Restoring Modified Static Base (SB) register

- Do not modified the FB register by the asm functions, because which use for the stack flame pointer.

c. Notes on Labels

The assembler source files generated by NC30 include internal labels in the format shown in Figure B.39. Therefore, you should avoid using labels in an asm function that might result in duplicate names.

```
Labels consisting of one uppercase letter and one or more numerals:

          A1:
          C9830:

Labels consisting of two or more characters preceded by the underscore (_):

          __LABEL:
          ___START:
```

Figure B.39 Label Format Prohibited in asm Function

B.3 Description of Japanese Characters

NC30 allows you to include Japanese characters in your C source programs. This chapter describes how to do so.

B.3.1 Overview of Japanese Characters

In contrast to the letters in the alphabet and other characters represented using one byte, Japanese characters require two bytes. NC30 allows such 2-byte characters to be used in character strings, character constants, and comments. The following character types can be included:

- kanji
- hiragana
- full-size katakana
- half-size katakana

Only the following kanji code systems can be used for Japanese characters in NC30.

- EUC (excluding user-defined characters made up of 3-byte code)
- Shift JIS (SJIS)

The character code for wide characters in C++ is UCS2.

B.3.2 Settings Required for Using Japanese Characters

The following environment variables must be set in order to use kanji codes. default specifies:

- Environment variable specifying input code system ....NCKIN
- Environment variable specifying output code system ....NCKOUT

Figure B.40 is an example of setting the environment variables.

```
Include the following in your autoexec.bat file:

How to input and output Shift JIS.
        set NCKIN = SJIS
        set NCKOUT = SJIS

How to input ECU and output Shift JIS.
        set NCKIN = EUC
        set NCKOUT = SJIS
```

Figure B.40 Example Setting of Environment Variables NCKIN and NCKOUT

In NC30, the input kanji codes are processed by the cpp30 preprocessor. cpp30 changes the codes to EUC codes. In the last stage of token analysis in the ccom30 compiler, the EUC codes are then converted for output as specified in the environment variable.

B.3.3 Japanese Characters in Character Strings

Figure B.41 shows the format for including Japanese characters in character strings.

```
L"漢字文字列"
```

Figure B.41 Format of Kanji code Description in Character Strings

If you write Japanese using the format L"漢字文字列" as with normal character strings, it is processed as a pointer type to a char type when manipulating the character string. You therefore cannot manipulate them as 2-byte characters.

To process the Japanese as 2-byte characters, precede the character string with L and process it as a pointer type to a wchar_t type. wchar_t types are defined (typedef) as unsigned short types in the standard header file stdlib.h.

Figure B.42 shows an example of a Japanese character string.

```
#include <stdlib.h>

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

            (remainder omitted)
                  :
```

Figure B.42 Example of Japanese Character Strings Description

Figure B.43 is a memory map of the character string initialized in [1] in Figure B.42.

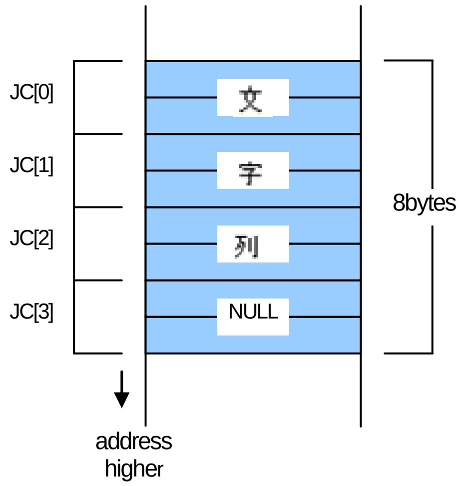


Figure B.43 Memory Location of wchar_t Type Character Strings

B.3.4 sing Japanese Characters as Character Constants

Figure B.44 shows the format for using Japanese characters as character constants.

```
L' 漢 '
```

Figure B.44 Format of Kanji code Description in Character Strings

As with character strings, precede the character constant with L and process it as a wchar_t type. If, as in ' 文字 ', you use two or more characters as the character constant, only the first character " 文 " becomes the character constant. Figure B.45 shows examples of how to write Japanese character constants.

```
#include <stdlib.h>

void        func(void)
{
            wchar_t     JC[5];

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

            (remainder omitted)
                :
```

Figure B.45 Format of Kanji Character Constant Description

Figure B.46 is a memory map of the array to which the character constant in Figure B.45 has been assigned.

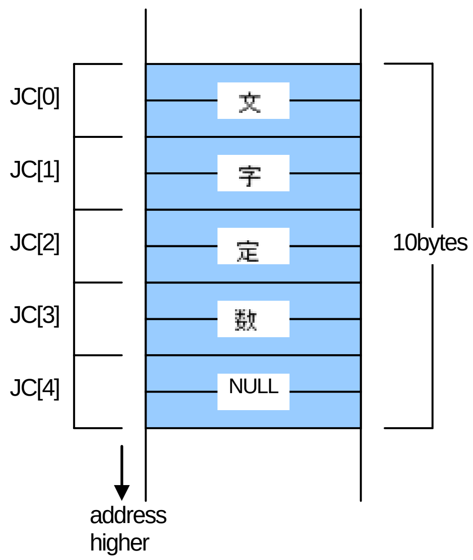


Figure B.46 Memory Location of wchar_t Type Character Constant Assigned Array

B.4 Default Argument Declaration of Function

NC30 allows you to define default values for the arguments of functions in the same way as with the C++ facility. This chapter describes NC30's facility to declare the default arguments of functions.
This feature is an extended specification provided for compiling in C mode. When in C++ mode, C++ language specifications apply.

B.4.1 Overview of Default Argument Declaration of Function

NC30 makes it possible to use implicit arguments by assigning default values to parameters when declaring prototypes for functions. Use of this feature saves time and effort to write the frequently used values when calling functions.

B.4.2 Format of Default Argument Declaration of Function

Figure B.47 shows the format used to declare the default arguments of a function.

```
Storage class specifier△Type declarator△Declarator([Dummy argument[=Default value or variable],...]);
```

Figure B.47 Format for declaring the default arguments of a function

An example for declaring a default parameter is shown in Figure B.48. The compilation result of the sample program in Figure B.48 is shown in Figure B.49.

```
int     func( int i=1 , int j=2 );      ← Declares the default values of parameters in the arguments to
                                          the function func as first argument: 1 and second argument: 2.

void    main(void)
{
        func();         ← The actual argument consists of the first argument: 1 and the second argument: 2.
        func(3);        ← The actual argument consists of the first argument: 3 and the second argument: 2.
        func(3,5);      ← The actual argument consists of the first argument: 3 and the second argument: 5.
}
```

Figure B.48 Example for declaring the default arguments of a function

```
;## # C_SRC :        {
        .glb         _main
_main:
        ._line       5
;## # C_SRC :                    func();
        mov.w        #0002H,R2             ← second argument :2
        mov.w        #0001H,R1             ← first argument     :1
        jsr          $func
        ._line       6
;## # C_SRC :                    func(3);
        mov.w        #0002H,R2             ← second argument :2
        mov.w        #0003H,R1             ← first argument     :3
        jsr          $func
        ._line       7
;## # C_SRC :                    func(3,5);
        mov.w        #0005H,R2             ← second argument :5
        mov.w        #0003H,R1             ← first argument     :3
        jsr          $func
        ._line       8
;## # C_SRC :        }
        rts
          :
        (omitted)
          :

Note) In NC30, arguments are stacked in revere order beginning with the argument that is declared last in the function.
In this example, arguments are passed via registers as they are processed.
```

Figure B.49 Compiling Result of smp1.c(smp1.a30)

A variable can be written for the argument of a function.
An example for specifying a variable for a default parameter is shown in Figure B.50. The compilation result of the sample program in Figure B.50 is shown in Figure B.51.

```
int     near sym ;
int     func( int i = sym);      ← Default argument is specified with a variable.

void    main(void)
{
        func();                  ← Function is called using variable (sym) as argument.
}
          :
        (omitted)
          :
```

Figure B.50 Example for specifying default argument with a variable (smp2.c)

```
_main:
          ._line 6
          mov.w     _sym,R1    ← Function is called using variable (sym) as argument.
          jsr       $func
          ._line 7
          rts
```

Figure B.51 Compile Result of smp2.c (smp2.a30)

B.4.3 Restrictions on Default Argument Declaration of Function

The default argument declaration of a function is subject to some restrictions as listed below. These restrictions must be observed.

a. When specifying a default value for multiple arguments

When specifying a default value in a function that has multiple arguments, always be sure to write values beginning with the last argument. Figure B.52 shows examples of incorrect description.

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

Figure B.52 Example for Writing a Function Prototype Declaration

b. When specifying a variable for a default value

To specify a variable as default value, declare the variable to specify before declaring a function prototype. If an undeclared variable is specified for the default value of a parameter as of the time the function prototype is declared, such specification is processed as an error.

B.5 inline Function Declaration

NC30 allows you to specify the inline storage class in the similar manner as in C++. By specifying the inline storage class for a function, you can expand the function inline.
This feature is an extended specification provided for compiling in C mode. When in C++ mode, C++ language specifications apply.

B.5.1 Overview of inline Storage Class

The inline storage class specifier declares that the specified function is a function to be expanded inline. The inline storage-class specifier indicates to a function that the function declared with it is to be expanded in-line. The functions specified as inline storage class have codes embedded directly in them at the assembly level.

B.5.2 Declaration Format of inline Storage Class

The inline storage class specifier must be written in a syntactically similar format to that of the static and extern-type storage class specifiers when declaring the inline storage class. Figure B.53 shows the format used to declare the inline storage class.

```
inline△specifier△function;
```

Figure B.53 Declaration Format of inline Storage Class

An example of a function declaration is shown in Figure B.54. The compilation result is shown in Figure B.55.

```
inline int    func(int i)              ← Prototype declaration of function
{
              return i++;
}

void          main(void)
{
              int        s;

              s = func(s);             ← Definition of body of function
}
```

Figure B.54 Sample program for inline functions (sample.c)

```
            .SECTION  program,CODE,ALIGN
            ._file    'sample.c'
            ._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++;          ← Inline storage class have codes
            mov.w     R0,R1                                embedded directly
            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:
            .align
            .END
```

Figure B.55 Compile Result of sample program (smp.a30)

B.5.3 Restrictions on inline Storage Class

When specifying the inline storage class, pay attention to the following :

(1) Regarding the indirect call of inline functions
The indirect call of an in line function cannot be carried out.It becomes a compile error when a indirect call is described.

(2) Regarding the recursive call of inline functions
The recursive call of an in line function cannot be carried out.It becomes a compile error when a recursive call is described.

(3) Regarding the definition of an inline function
When specifying inline storage class for a function, be sure to define the body of the function before calling it. Make sure that this body definition is written in the same file as the function is written . The description in Figure B.56 is processed as an error in NC30.

```
inline void   func(int i);

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

Error Message:
sample.c(5) : C2567 (E) inline function's body is not declared previously
===> func(1);

Figure B.56　Example of inappropriate code of inline function (1)

Furthermore, after using some function as an ordinary function if you define that function as an inline function later, NC30 becomes an error. (See Figure B.57.)

```
int           func(int i);

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

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

Error Message:
sample.c(9) : C2565 (E) inline function is called as normal function before
===>{

Figure B.57　Example of inappropriate code of inline function (2)

(4)　Regarding the address of an inline function

The inline function itself does not have an address. Therefore, if the & operator is used for an inline function, the software assumes an error. Figure B.58

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

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

               f = &func;
}
```

Error Message:

sample.c(10) : C2555 (E) can't get inline function's address by '&' operator
===> f = &func;

Figure B.58　Example of inappropriate code of inline function (3)

(5)　Declaration of static data

If static data is declared in an inline function, the body of the declared static data is allocated in units of files. For this reason, if an inline function consists of two or more files, this results in accessing different areas. Therefore, if there is static data you want to be used in an inline function, declare it outside the function. If a static declaration is found in an inline function, NC30 generates a warning. Renesas does not recommend entering static declarations in an inline function. Figure B.59

```
inline int     func( int j)
{
               static int   i = 0;

               i++;
               return i + j;
}
```

Warning Message:

sample.c(3) : C1636 (W) static variable in inline function
===> static int　　　i = 0;

Figure B.59　Example of inappropriate code of inline function (4)

(6)　Regarding debug information

NC30 does not output C language-level debug information for inline functions. Therefore, you need to debug inline functions at the assembly language level.

B.6 #pragma Extended Functions

B.6.1 Index of #pragma Extended Functions

Following index tables show contents and formation for #pragma extended functions.

a. Using Memory Mapping Extended Functions

Table B.4 Memory Mapping Extended Functions

Extented function	Description
#pragma BIT	Declares that the specified variable is an external variable present in an area where absolute bit instruction addressing is usable (i.e., variables present in an area from the address 00000H to the address 01FFFH). Syntax : #pragma BIT△variable name Example : #pragma BIT sym
#pragma SBDATA	Declares that the data uses SB relative addressing. Syntax : #pragma SBDATA△variable name Example : #pragma SBDATA sym
#pragma SECTION	Changes the section name generated by NC30 Syntax : #pragma SECTION△section_name△new_section_name Example : #pragma SECTION bss nonval_data
#pragma STRUCT	(1) Inhibits the packing of structures with the specified tag Syntax : #pragma STRUCT△structure_tag△unpack Example : #pragma STRUCT TAG1 unpack (2) Arranges members of structures with the specified tag and maps even sized members first Syntax : #pragma STRUCT△structure_tag△arrange Example : #pragma STRUCT TAG1 arrange ● This feature of "arrange" is effective only when compiling the source as a C program.
#pragma EXT4MPTR	A functional extension which shows a variable is a pointer accessing 4-Mbyte expanded space ROM. Syntax : #pragma EXT4MPTR△pointer variable name Example : #pragma EXT4MPTR sym
_ext4mptr	A functional extension which shows a variable is a pointer accessing 4-Mbyte expanded space ROM. Syntax : _ext4mptr△far△pointer variable declaration Example : _ext4mptr far int * ptr; ● This feature is effective only when compiling the source as a C program.

b. Using Extended Functions for Target Devices

Table B.5 Extended Functions for Use with Target Devices (1/2)

Extended function	Description
#pragma ADDRESS	Specifies the absolute address of a variable. For near variables, this specifies the address within the bank. Syntax : #pragma ADDRESS△variable-name△absolute-address Example : #pragma ADDRESS port0 2H
#pragma BITADDRESS	A variable is assigned to the bit position which the specified absolute address specified. Syntax: #pragma BITADDRESS△variable-name△bit-position, absolute-address Example : #pragma BITADDRESS io 1, 100H
#pragma INTCALL	Declares a function to invoke a software interrupt (int instruction). Syntax1 : #pragma INTCALL△INT number△assembler function name (registe-name) Example1 : #pragma INTCALL 25 func(R0, R1) Syntax2 : #pragma INTCALL△INT number△C language function name() Example2 : #pragma INTCALL 25 func() ● The parentheses can be omitted when register names are nonexistent.
#pragma INTERRUPT	Declares an interrupt processing function. By this declaration the compiler generates code to perform a procedure for an interrupt processing function on entry and exit to and from the function. Syntax : #pragma INTERRUPT△[/B|/E]△interrupt processing function name #pragma INTERRUPT△[/B|/E]△interrupt vector number△interrupt processing function name #pragma INTERRUPT△[/B|/E]△interrupt processing function name (vect=interrupt vector number) Example : #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)

Table B.6 Extended Functions for Use with Target Devices (2/2)

Extended function	Description
#pragma PARAMETER	Declares that when calling a function written in assembly language, the arguments to the function be passed via a register. Syntax : #pragma PARAMETER△function name (register name) Example : #pragma PARAMETER asm_func(R0, R1)
#pragma SPECIAL	Declares special page subroutine call functions. Syntax : #pragma SPECIAL△number△function-name() #pragma SPECIAL△function-name(vect=number) Example : #pragma SPECIAL 30 func() #pragma SPECIAL func() (vect=30) ● The parentheses following the function name can be omitted.

c. The Other Extended Functions

Table B.7　　The other extended functions

Extended function	Description
#pragma __ASMMACRO	Declares defined a function by assembler macro. Syntax： 　#pragma __ASMMACRO△function-name(register name) Example： 　#pragma　__ASMMACRO　mul(R0,R2)
#pragma ASM #pragma ENDASM	Specifies an area in which statements are written in assembly language. Syntax： 　#pragma△ASM 　#pragma△ENDASM Example： 　#pragma　ASM 　　mov.w　R0,R1 　　add.w　#02H,R1 　#pragma　ENDASM
#pragma PAGE	Specifies a page break for an assembler list file. Syntax： 　#pragma△PAGE Example： 　#pragma PAGE ● This feature is effective only when compiling the source as a C program.

Note that if a C++ language overloaded function is specified with #pragma, the nearest function written beneath the #pragma declaration is the subject that applies.
If an off-spec string or qualifier is written following #pragma, specification of how to process it is ignored.
Also, if an unsupported #pragma is used, warnings are, by default, not output. Warnings are output only when the -Wunknown_pragma option is specified.

When variable or function names are written using #pragma, it is possible to write qualified names.

```
#pragma ADDRESS s1              100H
unsigned short s1;
#pragma ADDRESS N::s2           200H
namespace N {
        unsigned short s2;
}
```

The functions that have "this" pointer are outside the scope of application of the relevant #pragma.

d. Extended features used for the C startup

These pragma's are used exclusively for the C startup. Therefore, do not use them in user programs.

Extended function	Description
#pragma STACKSIZE	Outputs a stack section (stack) and generates the top label name of the stack.
#pragma ISTACKSIZE	Outputs an interrupt stack section (istack) and generates the top label name of the interrupt stack.
#pragma CREG	When the internal register declared with this pragma is accessed, the compiler generates code to access it using a dedicated instruction.
#pragma sectaddress	Defines a section by the section name declared with this pragma. If an address is specified at the same time, the compiler outputs an address definition that uses the directive command ".org."
#pragma entry	Does not output the enter instruction to build a stack frame for the function declared with this pragma. This is to inhibit enter instructions from being generated before the stack pointer is initialized.
#pragma interrupt/V	Defines only an interrupt vector for the function declared with this pragma.

B.6.2 Using Memory Mapping Extended Functions

NC30 includes the following memory mapping extended functions.

#pragma BIT

SB Relative Addressing Using Variable Description Function

Function: Declares a variable present in an area where absolute bit instruction addressing is usable.

Syntax: #pragma BIT△variable_name

Description: For the M16C series and R8C family, ROM-efficient absolute bit instruction addressing can be used for variables present in an area from the address 00000H to the address 01FFFH.
The variables declared with #pragma BIT are assumed to be present in an area where absolute bit instruction addressing is usable.

Rules:
(1) If #pragma BIT is used for anything other than an external variable, it is ignored as invalid.
(2) If the variables declared with #pragma BIT cannot use absolute bit instruction addressing, the compiler uses address register indirect bit instruction addressing.
(3) Write this declaration before the variable is declared.

Example:

```
#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;
                :
                (omitted)
                :
```

Figure B.60 Example Use of #pragma BIT Declaration

Note: The instructions that use absolute bit instruction addressing are generated when the following conditions apply.
(1) When a -fbit(-fB) option is specified and the object to be operated on is a near-type variable
(2) When the object to be operated on is a variable declared by #pragma SBDATA
(3) When the object to be operated on is a variable declared by #pragma ADDRESS and the variable is located somewhere between address 0000H to address 01FFFH
(4) When the object to be operated on is a variable declared by #pragma BIT
(5) Variables mapped to areas within 32 bytes of the value of the FB register.

#pragma SBDATA

SB Relative Addressing Using Variable Description Function

Function: Declares that the data uses SB relative addressing.

Syntax: #pragma SBDATA△valuable-name

Description: For the M16C series and R8C family, use of SB relative addressing makes it possible to select efficient instructions. A #pragma SBDATA declares that SB relative addressing be used when referencing data for variables. This feature allows the compiler to generate ROM-efficient code.

Rules:

(1) The variable which has had #pragma SBDATA declared is declared with the assembler directive ".SBSYM."
(2) If #pragma SBDATA is specified for anything other than a variable, it is ignored as invalid.
(3) If the specified variable is a static variable declared in a function, the #pragma SBDATA declaration is ignored as invalid.
(4) The variable declared to be #pragma SBDATA is placed in a SBDATA attribute section when allocating memory for it.
(5) If #pragma SBDATA is declared for ROM data, the data is not placed in a SBDATA attribute section
(6) When the -fauto_over_255 (-FAO2) option is specified, the #pragma SBDATA declaration has no effect and a warning message "compile option -fauto_over_255 is specified, #pragma SBDATA was ignored" is output.
(7) Write this declaration before the variable is declared.

Example:

```
#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;
             :
          (omitted)
             :
```

Figure B.61 Example Use of #pragma SBDATA Declaration

Note: NC30 is premised on an assumption that the SB register will be initialized after reset and will thereafter be used as a fixed quantity.

#pragma SECTION

Change section name

Function : Changes the names of sections generated by NC30

Syntax : #pragma SECTION△section name△new section nam

Description : Specifying the program section, data section and rom section in a #pragma SECTION declaration changes the section names of all subsequent functions.
Specifying a bss section in a #pragma SECTION declaration changes the names of all data sections defined in that file.
If you need to add or change section names after using this function to change section names, change initialization, etc., in the startup program for the respective sections.
The default sections changeable with #pragma SECTION are only four—program, rom, data, and bss.

Example :

```
C source program:

#pragma   SECTION    program    pro1        ← Changes name of program section to pro1
void       func( void );
                  :
           (remainder omitted)


Assembler source program:

;### FUNCTION func
           .section      pro1               ← Maps to pro1 section
           ._file        'smp.c'
           ._line        9
           .glb          _func
_func:


Change name of data section from data to data1:

#pragma   SECTION   data   data1
int        i = 0;                           ← Maps to data1_NE section

void       func(void)
{
           (remainder omitted)
}

#pragma   SECTION   data   data2
int        j =1;                            ← Maps to data2_NE section */

void       sub(void)
{
           (remainder omitted)}
}
```

Figure B.62 Example Use of #pragma SECTION Declaration

Supplement: When modifying the name of a section, note that the section's location attribute (e.g., _NE or _NEI) is added after the section name.
If any string other than program, data, rom, bss, or interrupt is used as a default section name, the compiler outputs a warning and ignores this pragma line.

#pragma SECTION

Change section name

Note : In this compiler V.3.10 or earlier, the data and rom sections, as with the bss section, could only have their names altered in file units. For this reason, the programs created with V.3.10 or earlier require paying attention to the position where #PRAGMA SECTION is written. String data is output with the rom section name that is last declared.
When a string other than program, data, rom, and bss is specified as a section name, NC30 outputs a warning message and ignores this #pragma statement.

#pragma STRUCT

Control structure mapping

Function :

(1) Inhibits packing of structures
(2) Arranges structure members

Syntax :

(1) #pragma STRUCT△structure_tag△unpack
(2) #pragma STRUCT△ structure_tag△arrange (effective for only C)

Description and Examples :

In NC30, structures are packed by default. For example, the size of the structure in Figure B.63 is an odd number but there is no padding at the end of the structure for alignment.
When alignment is required, use #pragma STRUCT unpack to declare the structure. Members of the structure are always packed and, without any padding, arranged in the order they were declared.
Instead of padding, use #pragma STRUCT arrange to arrange the order of members so that the structure will be aligned.

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

Member name	Type	Size	Mapped location (offset)
i	int	16bits	0
c	char	8bits	2
j	int	16bits	3

Figure B.63 Example Mapping of Structure Members (1)

Rules :

(1) Inhibiting packing of structures
This NC30 extended function allows you to control the alignment of the structure. Shown in Figure B.64 is an example where the structure in Figure B.63 is inhibited from being packed with STRUCT.

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

Member name	Type	Size	Mapped location (offset)
i	int	16bits	0
c	char	8bits	2
j	int	16bits	3
Padding	(char)	8bits	-

Figure B.64 Example Mapping of Structure Members (2)

As shown Figure B.64, if the total size of the structure members is an odd number of bytes, #pragma STRUCT adds 1 byte as packing after the last member. Therefore, if you use #pragma STRUCT to inhibit padding, all structures have an even byte size.

#pragma STRUCT

Control structure mapping

Rules :

(2) Arranging members

This NC30 extended function allows you to map the all even-sized structure members first, followed by odd-sized members. Shown in Figure B.65 is an example where an arrangement of the structure in Figure B.63 is rearranged with #pragma STRUCT.

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

Member name	Type	Size	Mapped location (offset)
i	int	16bits	0
j	int	16bits	2
c	char	8bits	4

Figure B.65 Example Mapping of Structure Members (3)

You must declare #pragma STRUCT for inhibiting packing and arranging the structure members before defining the structure members.

(3) Template class cannot be specified for *structure_tag*.

Examples :

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

Figure B.66 Example of #pragma STRUCT Declaration

Supplement: This feature of "arrange" is effective only when compiling the source as a C program.

Note : If the word "unpack" or "arrange" is not written, the compiler outputs a warning. In that case, this #pragma specification has no effect.

#pragma EXT4MPTR

denition a data allocated on 4 Mbyte extension space ROM area

Function : A functional extension which shows a variable is a pointer accessing 4-Mbyte expanded space ROM.

Syntax : #pragma EXT4MPTR△pointer_name

Description : His feature is provided for extension mode 2(4M bytes extension mode) which is available with some products in the M16C/62 group.
Declare a pointer variable for accessing a 4M bytes space. When so declared, the compiler generates code for switching banks as necessary to access a 4M bytes space.
This bank-switching code is generated one for each function in the place where the pointer is used first. In successive operations, therefore, the banks are set only once.
When using multiple pointer variables, use the "-fchange_bank_always (-fCBA)" option which sets the banks each time the program accesses the 4M bytes space.

Rules :

(1) If, while the option -fchange_bank_always (-fCBA) is not specified, #pragma EXT4MPTR is written twice or more in one translation unit, the compiler outputs an error message "multiple #pragma EXT4MPTR's pointer."

(2) If #pragma EXT4MPTR is followed by only white-space characters or a pointer variable name is specified with invalid characters (e.g., variable name "1234" is improbable), the compiler outputs a warning "#pragma EXT4MPTR format error, ignored" and ignores this line.

(3) This feature is effective only when compiling the source as a C program.

Examples :

```
C source program:
        #pragma EXT4MPTR pointer
        struct tagh{
                int bitmap;
                char code;
        } far *pointer;
        void main(void)
        {
                int data;
                data = pointer->bitmap;
        }
Assembly language source program:
        mov.w   _pointer,A0
        mov.w   _pointer+2,A1
        .glb    __BankSelect
        mov.b   A1,__BankSelect    <- Change the bank
        bclr    3,A1
        bset    2,A1
        lde.w   [A1A0],-2[FB]
```

Figure B.67 Example Use of #pragma EXT4MPTR Declaration

Note :

(1) Before using this feature, check to see if the microcomputer and the system (hardware) support 4M bytes extension space mode.

(2) If the option -R8C and -R8CE are used, this declaration is ignored.

(3) Write this declaration before the variable is declared.

_ext4mptr

denition a data allocated on 4 Mbyte extension space ROM area

Function : A functional extension which shows a variable is a pointer accessing 4-Mbyte expanded space ROM.

Syntax : _ext4mptr far△pointer variable declaration

Description : His feature is provided for extension mode 2 (4M byte extension mode) which is available with some products in the M16C/62 group.
Declare a pointer variable for accessing a 4M-byte space. When so declared, the compiler generates code for switching banks as necessary to access a 4M-byte space.
This bank-switching code is generated one for each function in the place where the pointer is used first. In successive operations, therefore, the banks are set only once.
When using multiple pointer variables, use the "-fchange_bank_always (-fCBA)" option which sets the banks each time the program accesses the 4M-byte space.

Rules : This feature is effective only when compiling the source as a C program.

Examples :

```
C source program:
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    ← Change the bank
        bclr 3,A1
        bset 2,A1
        lde.w [A1A0],-2[FB]
```

Figure B.68 Example Use of #pragma _ext4mptr Declaration

Note :
(1) Before using this feature, check to see if the microcomputer and the system (hardware) support 4M-byte extension space mode.
(2) If the option -R8C and -R8CE are used, this declaration is ignored.

B.6.3 Using Extended Functions for Target Devices

NC30 includes the following extended functions for target devices.

#pragma ADDRESS

Specify absolute address of I/O variable

Function : Specifies the absolute address of a variable. For near variables, the specified address is within the bank.

Syntax : #pragma ADDRESS△variable-name△absolute-address

Description : The absolute address specified by this declaration is unrolled as string into an assembler source file wherein it is defined with the assembler directive ".EQU." Therefore, the form in which to write numeric values depends on the assembler. Numeric representations in the assembler are shown below.

- Append 'B' or 'b' to binary numbers
- Append 'O' or 'o' to octal numbers
- Write decimal integers only.
- Append 'H' or 'h' to hexadecimal numbers. If the number starts with letters A to F, precede it with 0.

Rules :

(1) All storage classes such as extern and static for variables specified in #pragma ADDRESS are invalid.
(2) Variables specified in #pragma ADDRESS are valid only for variables defined outside the function.
(3) This directive must be issued before the variable is declared. In C-language compilation, however, the directive is also valid for a variable that has already been declared.
(4) #pragma ADDRESS is invalid if you specify other than a variable.
(5) In C-language compilation, the #pragma ADDRESS directive is also valid for a variable that was already declared. No error occurs if a #pragma ADDRESS declaration is duplicated, but the last declared address is valid.
(6) A warning occurs if you include an initialization expression and an initialization expression is invalid.
(7) Since #pragma ADDRESS is normally used for I/O variables, it is processed as having volatile specified, irrespective of the presence of volatile specification.
(8) The variables declared by a #pragma ADDRESS declaration cannot be externally referenced.
(9) If the option -fnot_address_volatile (-fNAV) is specified, the compiler does not handle the #pragma ADDRESS-specified variable as being specified as volatile.
(10) If #pragma ADDRESS is followed by only white-space characters or a variable name is specified with invalid characters (e.g., "123") or there are only white-space characters in the address part, the compiler outputs a warning "#pragma ADDRESS format error, ignored" and ignores this line.
(11) If the character string in the address *absolute-address* a character for which the 8th bit is 1, the warning message 'Kanji in #pragma ADDRESS' is output.

Examples :

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

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

Figure B.69 #pragma ADDRESS Declaration

#pragma BITADDRESS

The bit position specification absolute address allotment function of an input-and-output variable

Function : A variable is assigned to the bit position which the specified absolute address specified.

Syntax : #pragma BITADDRESS△variable-name△bit-position,absolute-address

Description : The absolute address specified by this declaration is unrolled as string into an assembler source file wherein it is defined with the assembler directive ".BITEQU." Therefore, the form in which to write numeric values depends on the assembler. Numeric representations in the assembler are shown below. Also, the writable range of bit positions is shown below.

(1) The bit position
- It is the range of 0-65535.Only the decimal digit.

(2) The Address
- Append 'B' or 'b' to binary numbers
- Append 'O' or 'o' to octal numbers
- Write decimal integers only.
- Append 'H' or 'h' to hexadecimal numbers. If the number starts with letters A to F, precede it with 0.

Rules :

(1) Only _Bool-type or bool-type variables can be specified for the variable name. Variables of other than _Bool and bool types, if specified, result in an error.
(2) All storage classes such as extern and static for variables specified in #pragma BITADDRESS are invalid.
(3) Variables specified in #pragma BITADDRESS are valid only for variables defined outside the function.
(4) This directive must be issued before the variable is declared. In C-language compilation, however, the directive is also valid for a variable that has already been declared.
(5) #pragma BITADDRESS is invalid if you specify other than a variable.
(6) Issuing a #pragma BITADDRESS directive twice for the same variable leads to an error.
(7) An error occurs if you include an initialization expression.
(8) Since #pragma BITADDRESS is normally used for I/O variables, it is processed as having volatile specified, irrespective of the presence of volatile specification.
(9) If the option -fnot_address_volatile (-fNAV) is specified, the compiler does not handle the #pragma ADDRESS-specified variable as being specified as volatile.
(10) If #pragma BITADDRESS is followed by only white-space characters or a variable name is specified with invalid characters (e.g., "123") or there are only white-space characters in the address part, the compiler outputs a warning "#pragma BITADDRESS format error, ignored" and ignores this line.
(11) If the string in the address part contains a character whose 8th bit = 1, the compiler outputs a warning "Kanji in #pragma ADDRESS" and ignores this line. Write this declaration before the variable is declared

Example :

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

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

Figure B.70 #pragma BITADDRESS Declaration

#pragma INTCALL

Declare a function called by the INT instruction

Function : Declares a function called by a software interrupt (by the int instruction).

Syntax :
(1) #pragma INTCALL△INT number△assembler function name (register name, register name, …)
(2) #pragma INTCALL△INT number△C function name

Description : The compiler issues the int instruction by a specified INT number and calls the function by a software interrupt.

Rules :

- Declaring assembler functions

(1) Write a #pragma INTCALL declaration before a prototype for the assembler function is declared. When compiled as a C program, it doesn't matter if #pragma INTCALL occurs after the prototype declaration.

(2) Observe the following for the function prototype declaration:

  (1) Make sure that the number of parameters in the prototype declaration matches those in the #pragma INTCALL declaration.
  (2) You cannot declare the following types in the parameters in the assembler function.
    - Structure type
    - union type
    - double type
    - long double type
    - long long type
  (3) You cannot declare the following functions as the return values of assembler functions.
    - Functions that return structures or unions

(3) To call the function, the following registers can be used as arguments.

- float types, long types (32-bit registers)
  R2R0 and R3R1
- far pointer types (32-bit registers)
  R2R0,R3R1, and A1A0
- int types, near pointer types (16-bit registers)
  A0,A1,R0,R1,R2, and R3
- char types, _Bool types and bool types (8-bit registers)
  R0L, R0H, R1L, and R1H
- There is no differentiation between uppercase and lowercase letters in register names.

(4) You can only use decimals for the INT Numbers.

- Declaring functions of which the body is written in C/C++.
  (1) Write a #pragma INTCALL declaration before function prototype is declared. When compiled as a C program, it doesn't matter if #pragma INTCALL occurs after the prototype declaration.
  (2) Observe the following in the prototype declaration:
    (1) In function prototype declarations, only a function for which all of arguments are passed via register, as in rules for function calls, can be declared.
    (2) Functions whose return types are structure or union types cannot be declared.
  (3) If there are no register names parentheses can be omitted. The parentheses following the C function name can be omitted.
  (4) INT numbers can only be written in decimal.
  (5) INT numbers can be specified in the range 0 to 63. Otherwise, an error results and a message "Invalid #pragma INTCALL interrupt number" is output.

#pragma INTCALL

Declare a function called by the INT instruction

Examples :

```
#pragma INTCALL 25 asm_func(R2R0, R1)
int asm_func(unsigned long, unsigned int);     ← Prototype declaration for an assembler function

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

            i = 0x7FFD;
            l = 0x007F;

            asm_func( l, i );                  ← Calling the assembler function
}
```

Figure B.71 Example of #pragma INTCALL Declaration(asm function) (1)

```
#pragma INTCALL 25 c_func();                   ←You may NOT specify registers
int c_func(unsigned int, unsigned int);        ←Prototype declaration for a C function

void        main(void)
{
            int         i, j;

            i = 0x7FFD;
            j = 0x007F;

            c_func( i, j );                    ← Calling the C function
}
```

Figure B.72 Example of #pragma INTCALL Declaration(C language functuion) (2)

Note: To use the startup file included with the product, alter the content of the vector section before use. For details on how to alter it, refer to " Chapter 2 Preparing the Startup Program."

#pragma INTERRUPT

Declare interrupt function

Function : Declares an interrupt handler

Syntax :

(1) #pragma INTERRUPT△[/B|/E]△interrupt-handler-name
(2) #pragmaINTERRUPT△[/B|/E]△interrupt-vector-number△interrupt-handler-name
(3) #pragmaINTERRUPT△[/B|/E]△interrupt-handler-name(vect=interrupt-vector-number)

Description :

(1) When an interrupt handling function is declared in the form shown above, the compiler generates code to perform the following interrupt servicing process on entry and exit to and from the function.
- In entry processing, all registers of the Micro Procesor are saved to the stack.
- In exit processing, the saved registers are restored and control is returned to the calling function by the REIT instruction.

(2) You may specify either /B or /E in this declaration
- [/B]
  Instead of saving the registers to the stack when calling the function, you can switch to the alternate registers. This allows for faster interrupt processing.
  To use the back register, make sure that the back register is not altered by nesting of interrupts.
- [/E]
  Enables multiple-interrupts (i.e., interrupt from within another) immediately after entering an interrupt. This results in an increased interrupt response.

(3) Interrupt vector numbers can be specified at declaration time.

(4) When an interrupt vector number is written, the compiler automatically generates a variable vector table.
- The variable vector table is generated in an object file for each #pragma INTERRUPT-declared file.
- The generated variable vector table can be verified by checking the map file generated by optlnk.

Note that the automatically generated variable vector table generates a section "vector." Be aware that if there is any vector section in the program, a link error occurs.

(5) Write this declaration before the function prototype is declared.

#pragma INTERRUPT

Declare interrupt function

Rules :

(1) A warning is output when compiling if you declare interrupt processing functions that take parameters
(2) A warning is output when compiling if you declare interrupt processing functions that return a value. Be sure to declare that any return value of the function has the void type.
(3) Only functions for which the function is defined after a #pragma INTERRUPT declaration are valid.
(4) No processing occurs if you specify other than a function name.
(5) No error occurs if you duplicate #pragma INTERRUPT declarations.
(6) If switches /E and /B are specified at the same time, the compiler outputs a warning message "#pragma INTERRUPT conflict, ignored" and ignores this line.
(7) If different vector numbers are written in the same interrupt handling function, the vector number that is declared last has priority.

```
#pragma   INTTERUPT   intr(vect=10)
#pragma   INTTERUPT   intr(vect=20)          /* The interrupt vector number 20 is effective. */
```

Figure B.73 Example for writing different interrupt vector numbers

(8) A compile warning occurs if you use any function specified in one of the following declarations in #pragma INTERRUPT:
- #pragma ALMHANDLER
- #pragma INTHANDLER
- #pragma HANDLER
- #pragma CYCHANDLER
- #pragma TASK

(9) If #pragma INTERRUPT is followed by only white-space characters or the interrupt handling function name contains an invalid string (e.g., "123"), the compiler outputs a warning "#pragma INTERRUPT format error, ignored" and ignores this line.
(10) If any number other than 0–63 is written as a vector number, the compiler outputs a warning "Invalid, #pragma INTERRUPT vector number" and ignores this line.
(11) Write this declaration before the function prototype is declared.

Example :

```
extern int    int_counter;

#pragma   INTERRUPT   /B   i_func

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

Figure B.74 Example of #pragma INTERRUPT Declaration

Note :

To use a #pragma INTERRUPT that has no interrupt vector numbers written, it is necessary to define a section "vector." For details on how to change, see Chapter 2, "Preparing the Startup Program."

#pragma PARAMETER

Declare assembler function that passed arguments via register

Function : Declares an assembler function that passes parameters via registers

Syntax : #pragma PARAMETER△assembler-function-name(register-name,register-name,…)

Description : This extended function declares that, when calling an assembler function, its parameters are passed via registers.

- float types, long types (32-bit registers)
  R2R0 and R3R1
- far pointer types (32-bit registers)
  R2R0, R3R1,and A1A0
- int types, near pointer types (16-bit registers)
  A0, A1, R0, R1, R2, and R3
- char types and _Bool types (8-bit registers)
  R0L, R0H, R1L, and R1H
- Register names are NOT case-sensitive.
- Long long type (64-bit integer type) and double type, as well as structure and union types cannot be declared.

Rules :

(1) Write a #pragma PARAMETER declaration before a prototype for the assembler function is declared. When compiled as a C program, it doesn't matter if #pragma PARAMETER occurs after the prototype declaration.

(2) When writing a prototype declaration, observe the following:

  (1) It is necessary that the number of parameters in the prototype declaration and those in the #pragma PARAMETER declaration should match.

  (2) The following types cannot be declared as parameters for an assembler function in a #pragma PARAMETER declaration:
  - structure type
  - union type
  - double type
  - long double type
  - long long type

  (3) The assembler functions shown below cannot be declared:
  - Functions returning structure or union type

(3) The output symbols of the function specified with #pragma PARAMETER always have an underscore (_) added.

Example :

```
#pragma PARAMETER asm_func(R0, R1)
int asm_func(unsigned int, unsigned int);        ←Prototype declaration for an assembler function
void        main(void)
{
            int         i, j;

            i = 0x7FFD;
            j = 0x007F;

            asm_func( i, j );                    ← Calling the assembler function
}
```

Figure B.75 Example of #pragma PARAMETER Declaration

#pragma SPECIAL

Declare a special page subroutine call function

Function : Declares a special page subroutine call (JSRS instruction) function

Syntax :
(1) #pragma SPECIA△call number△function-name()
(2) #pragma SPECIAL△function-name(vect = call number)

Description :
(1) The function declared with #pragma SPECIAL is assumed to be located at the address set in the special page vector table plus 0F0000H, and special page subroutines are called as such.
(2) A special page vector table can be automatically generated at link time.

Rules :
(1) Functions declared using #pragma SPECIAL are mapped to the program_S section. Be sure to map the program_S section between 0F0000H and 0FFFFFH.
(2) Calls are numbered between 18 and 255 in decimal only.
(3) An error occurs if different call numbers are recorded for the same function.

```
#pragma   SPECIAL   func(vect=20)
#pragma   SPECIAL   func(vect=30)             // Call number 30 is effective
```

Figure B.76 Example for writing different call numbers

(4) If functions are defined in one file and function calls are defined in another file, be sure to write this declaration in both files.
(5) The parentheses following the function name in form (1) can be omitted.
(6) The word "vect" in (vect = special page number) consists entirely of lowercase letters.
(7) Write this declaration before the function prototype is declared. When compiled as a C program, it doesn't matter if #pragma SPECIAL occurs after the prototype declaration.

Example :

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

void        main(void)
{
            int i, j;

            i = 0x7FFD;
            j = 0x007F;

            func( i, j );              ← special page subroutine call
}
```

Figure B.77 Example of #pragma SPECIAL Declaration

Note : If the function specified with #pragma SPECIAL is already specified with another #pragma, a compile error results.

B.6.4 The Other Extensions

NC30 includes the following extended function.

#pragma __ASMMACRO

Assembler macro function

Function : Declares defined a function by assembler macro.

Syntax : #pragma __ASMMACRO . function-name(register name, …)

Rules :

(1) Write this declaration before a prototype for the function is declared. When compiled as a C program, it doesn't matter if #pragma __ASMMACRO occurs after the prototype declaration. Be sure that assembler macro functions are declared as static. If static declarations are nonexistent, an assembler error results.

(2) Can't declare the function of no parameter. Parameter is passed via register.Please specify the register matching the parameter type.

(3) Please append the underscore ("_") to the head of the definition assembler macro name.

(4) The following is a return value-related calling rules. You can't declare structure and union type as the return value.

_Bool type, char type : R0L	float type : R2R0
int type, short type : R0	double type : R3R2R1R0
long type : R2R0	long long type : R3R1R2R0

(5) If you change the register's data, save the register to the stack in entry processing of assembler macro function and the saved register restore in exit processing.

(6) If #pragma __ASMMACRO is declared after a function call, the compiler outputs an error message "#pragma __ASMMACRO must be declared before use."

(7) If #pragma __ASMMACRO is declared for an identifier that is not a function, the compiler outputs a warning message "#pragma __ASMMACRO not function, ignored" and ignores this pragma.

(8) Unless a function declaration is written in the prototype declaration form, the compiler outputs a warning message "#pragma __ASMMACRO's function must be prototyped, ignored" and ignores this pragma.

Example :

```
#pragma __ASMMACRO mul( R0, R2 )
static long mul( int, int );                    /* Be sure to declare "static" *//

asm("   _mul     .macro¥n"
        "   mul.w   R2,R0¥n"
        "   .endm");

long        l;

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

Figure B.78 Example of #pragma __AMMACRO

#pragma ASM, #pragma ENDASM

Inline assembling

Function : Specifies assembly code in C.

Syntax :

#pragma ASM
assembly statements
#pragma ENDASM

Description : Outputs a range of lines written between #pragma ASM and #pragma ENDASM to the assembler source file directly as are.
Writing #pragma ASM, be sure to use it in combination with #pragma ENDASM. this compiler suspends processing if no #pragma ENDASM is found the corresponding #pragma ASM.

Rules :

(1) In an assembly language description, do not write a statement that will change register contents. If such a statement needs to be written, use push and pop instructions to save and restore the register contents.
(2) Within the "#pragma ASM" to "#pragma ENDASM" section, do not reference arguments and auto variables.
(3) Within the "#pragma ASM" to "#pragma ENDASM" section, do not write a branch statement (including conditional branch) which may affect the program flow.
(4) The symbols that begin with '$' or '_' are the reserved symbols for the compiler. Behavior of a program where a definition of symbols beginning with '$' or '_' is written within #pragma asm to endasm cannot be guaranteed.
(5) Do not write the directive ".section" within #pragma asm to endasm. To change a section name, be sure to use #pragma SECTION outside the range #pragma asm to endasm.
(6) If #pragma ASM is followed by other than white-space characters, the compiler outputs a warning and ignores the #pragma ASM line. As a result, the line next to #pragma ASM is interpreted as a C source.
(7) If, while #pragma ASM is written but #pragma ENDASM is nonexistent, the end of the file being loaded is reached, the compiler displays a fatal error "no #pragma ENDASM."
(8) If one line of assembly language description including new-line code exceeds 1,024 characters, the compiler outputs a warning "#pragma ASM line too long, then cut" and ignores a range of characters from the 1,024th and on to new-line code.
(9) If a line in assembly language includes the comment-opening character (;), the compiler converts the whole line into output kanji code if this matches the setting of environment variable NCKOUT. However, this only suits the setting of environment variable NCKIN when option -E or -P is specified.

#pragma ASM, #pragma ENDASM

Inline assembling

Example :

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

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

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

#pragma   ENDASM
}
```

This range of lines is output to the assembler source line directly as are.

Figure B.79 Example of #pragma ASM(ENDASM)

Suppliment : It is this assembly language program written between #pragma ASM and #pragma ENDASM that is processed by the C preprocessor.

#pragma PAGE

Output .PAGE

Function : Specifies a page break for an assembler list file.

Syntax : #pragma PAGE

Description : If #pragma PAGE is written in the source file, the compiler outputs the assembler directive ".PAGE" to the assembler list file it outputs. This feature makes it possible to specify page breaks when assembler list files are output by the assembler.

Rules :

(1) Strings specified in the header of the assembler directive ".PAGE" cannot be specified.
(2) You cannot write a #pragma PAGE in an auto variable declaration.
(3) This feature is effective only when compiling the source as a C program.

Example :

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

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

Figure B.80 Example of #pragma PAGE

B.7 assembler Macro Function

B.7.1 Outline of Assembler Macro Function

This compiler allows part of assembly language to be written as functions in C.
Because specific assembler commands can be written directly in a C-language program, you can easily tune up the program.

B.7.2 Description Example of Assembler Macro Function

Assembler macro functions can be written in a C-language program in the same format as C-language functions, as shown below.
If you use assembler macro function feature, please be sure to include your asmmacro.h.

```
#include  <asmmacro.h>              /* Includes the assembler macro function definition file */
long      l;
char      a[20];
char      b[20];

void      func(void)
{
          l = rmpa_b(0,19,a,b);     /* asm Macro Function(rmpa command) */
}
```

Figure B.81 Description Example of Assembler Macro Function

B.7.3 Commands that Can be Written by Assembler Macro Function

Shows assembly language writable in assembler macro functions and its functionality and form as an assembler macro function.

ABS

Function : Returns the absolute value of val.

Syntax :

```
#include <asmmacro.h>

static signed char abs_b( signed char val );
/* When calculated in 8 bits */

static signed int abs_w( signed int val );
/* When calculated in 16 bits */
```

DADC

Function : Returns the result of decimal addition with carry on val1 plus val2.

Syntax :

```
#include <asmmacro.h>

static unsigned char dadc_b(unsigned char val1,unsigned   char val2);
/* When calculated in 8 bits */

static unsigned int   dadc_w(unsigned int val1, unsigned int val2);
/* When calculated in 16 bits */
```

DADD

Function : Returns the result of decimal addition with no carry on val1 plus val2.

Syntax :

```
#include <asmmacro.h>

static unsigned char dadd_b(unsigned char val1, unsigned char val2);
/* When calculated in 8 bits */

static unsigned int   dadd_w(unsigned int val1, unsigned int val2);
/* When calculated in 16 bits */
```

DIV

Function : Returns the quotient of a division where the dividend val2 is divided by the divisor val1 with the sign included.

Syntax : #include <asmmacro.h>

```
static signed char div_b(signed char val1, signed int val2);
/* 16 bits divided by 8 bits with signed */

static signed int div_w(signed int val1, signed long val2);
/* 32 bits divided by 16 bits with signed */
```

DIVU

Function: Returns the quotient of a division where the dividend val2 is divided by the divisor val1 with the sign not included.

Syntax : #include <asmmacro.h>

```
unsigned char divu_b(unsigned char val1, unsigned int val2);
/* 16 bits divided by 8 bits with unsigned */

unsigned int divu_w(unsigned int val1, unsigned long val2);
/* 32 bits divided by 16 bits with unsigned */
```

DIVX

Function: Returns the quotient of a division where the dividend val2 is divided by the divisor val1 with the sign not included.

Syntax : #include <asmmacro.h>

```
static signed char divx_b( unsugned char val1, signed int val2 );
/* 16 bits divided by 8 bits with unsigned */

static signed int divx_w( signed int val1, signed long val2 );
/* 32 bits divided by 16 bits with unsigned */
```

MOD, MODU

Function: Devide val1 by val2 and get mod.

Syntax :

```
#include <asmmacro.h>

static signed char mod_b(signed char val1,signed int val2);
/* 16 bits divided by 8 bits with signed */

static signed int mod_w(signed int val1,signed long val2);
/* 32 bits divided by 16 bits with signed */

static unsigned char modu_b(unsigned char val1,unsigned int val2);
/* 16 bits divided by 8 bits with unsigned */

static unsigned int modu_w(unsigned int val1,unsigned long val2);
/* 32 bits divided by 16 bits with unsigned */
```

NOT

Function : Returns the value of the inverted val.

Syntax :

```
#include <asmmacro.h>

static signed char not_b(signed char val);
/* When calculated in 8 bits */

static signedd int not_w(signed int val);
/* When calculated in 16 bits */
```

NEG

Function : Returns the two's complement of val.

Syntax :

```
#include <asmmacro.h>

static signed char neg_b(signed char val);
/* When calculated in 8 bits */

static signed int neg_w(signed int val);
/* When calculated in 16 bits */
```

DSBB

Function : Returns the result of decimal subtraction with borrow on val2 minus val1.

Syntax : #include <asmmacro.h>

static unsigned char dsbb_b(unsigned char val1, unsigned char val2);
/* When calculated in 8 bits */

static unsigned int dsbb_w(unsigned int val1, unsigned int val2);
/* When calculated in 16 bits */

DSUB

Function : Returns the result of decimal subtraction with no borrow on val2 minus val1.

Syntax : #include <asmmacro.h>

static unsigned char dsub_b(unsigned char val1, unsigned char val2);
/* When calculated in 8 bits*/

static unsigned int dsub_w(unsigned int val1, unsigned int val2);
/* When calculated in 16 bits */

MOVdir

Function : transfer to val2 from val1 by nibble

Syntax : #include <asmmacro.h>

static unsigned char movll(unsigned char val1,unsigned char val2);
/* to low of val2 from high of val1 */

static unsigned char movlh(unsigned char val1,unsigned char val2);
/* to high of val2 from low of val1*/

static unsigned char movhl(unsigned char val1, unsigned char val2);
/* to low of val2 from high of val1 */

static unsigned char movhh(unsigned char val1,unsigned char val2);
/* to high of val2 from high of val1 */

RMPA

Function : Initial value: init; Number of times: count. The result is returned after performing a sum-of-products operation assuming p1 and P2 as the start addresses where multipliers are stored.

Syntax :

```
#include <asmmacro.h>

static int rmpa_b(signed int init, unsigned int count, signed char _near *p1, signed char _near *p2);
/* When calculated in 8 bits */

static long rmpa_w(signed long init, unsigned int count, signed int _near *p1, signed int _near *p2);
/* When calculated in 16 bits*/
```

SMOVF

Function : Strings are transferred from the source address indicated by p1 to the destination address indicated by p2 as many times as indicated by count in the address-incrementing direction.
There is no return value.

Syntax :

```
#include <asmmacro.h>

static void smovf_b(unsigned char _near *p1, unsigned _near char *p2, unsigned int count);
/* When calculated in 8 bits */

static void smovf_w(unsigned int _near *p1, unsigned _near int *p2, unsigned int count);
/* When calculated in 16 bits*/
```

SHA

Function : The value of val is returned after arithmetically shifting it as many times as indicated by count.

Syntax :

```
#include <asmmacro.h>

static unsigned char sha_b(signed char count, unsigned char val);
/* When calculated in 8 bits */

static unsigned int sha_w(signed char count, unsigned int val);
/* When calculated in 16 bits */

static unsigned long sha_l(signed char count, unsigned long val);
/* When calculated in 32 bits */
```

SHL

Function : The value of val is returned after logically shifting it as many times as indicated by count.

Syntax : #include <asmmacro.h>

static unsigned char shl_b(signed char count, unsigned char val);
/* When calculated in 8 bits */

static unsigned int shl_w(signed char count, unsigned int val);
/* When calculated in 16 bits */

static unsigned long shl_l(signed char count, unsigned long val);
/* When calculated in 32 bits */

SMOVB

Function : Strings are transferred from the source address indicated by p1 to the destination address indicated by p2 as many times as indicated by count in the addressdecrementing direction. There is no return value.

Syntax : #include <asmmacro.h>

static void smovb_b(unsigned char _near *p1, unsigned char _near *p2, unsigned int count);
/* When calculated in 8 bits */

static void smovb_w(unsigned int _near *p1, unsigned int _near *p2, unsigned int count);
/* When calculated in 16 bits */

SSTR

Function : Strings are stored using val as the data to store, p as the address to from val address which to transfer, and count as the number of times to transfer data. There is no return value.

Syntax : #include <asmmacro.h>

static void sstr_b(unsigned char val, unsigned char _near *p, unsigned int count);
/* When calculated in 8 bits */

static void sstr_w(unsigned int val, unsigned int _near *p, unsigned int count);
/* When calculated in 16 bits */

ROLC

Function : The value of val is returned after rotating it left by 1 bit including the C flag.

Syntax : #include <asmmacro.h>

static unsigned char rolc_b(unsigned char val1);
/* When calculated in 8 bits */

static unsigned int rolc_w(unsigned int val1);
/* When calculated in 16 bits*/

RORC

Function : The value of val is returned after rotating it right by 1 bit including the C flag.

Syntax : #include <asmmacro.h>

static unsigned char rorc_b(unsigned char val);
/* When calculated in 8 bits */

static unsigned int rorc_w(unsigned int val);
/* When calculated in 16 bits */

ROT

Function : The value of val is returned after rotating it as many times as indicated by count.

Syntax : #include <asmmacro.h>

static unsigned char rot_b(signed char count, unsigned char val);
/* When calculated in 8 bits */

static unsigned int rot_w(signed char count, unsigned int val);
/* When calculated in 16 bits */

Appendix C Translation Limits

Table C.1 lists the translation limits of the compiler.
When creating a source program, make sure it is created within the range of these translation limits.

Table C.1 Translation Limits of Compiler (1/2)

Item	Specification
Number of characters per line of source file	512 bytes (characters) including the new line code
Number of lines in source file	65535 max.
Maximum number of files that can be specified in NC30	No limit (Memory capacity dependence)
Maximum length of filename	Depends on operating system
Maximum number of macros that can be specified in nc30 command line option -D	No limit (Memory capacity dependence)
Maximum number of directories that can be specified in nc30 command line option -I	256max
Maximum number of parameters that can be specified in nc30 command line option -as30	No limit (Memory capacity dependence)
Maximum nesting levels of compound statements, iteration control structures, and selection control structures	No limit (Memory capacity dependence)
Maximum nesting levels in conditional compiling	No limit (Memory capacity dependence)
Number of pointers modifying declared basic types, arrays, and function declarators	No limit (Memory capacity dependence)
Number of function definitions	No limit (Memory capacity dependence)
Number of identifiers with block scope in one block	No limit (Memory capacity dependence)
Maximum number of macro identifiers that can be simultaneously defined in one source file	No limit (Memory capacity dependence)
Maximum number of macro name replacements	No limit (Memory capacity dependence)
Number of logical source lines in input program	No limit (Memory capacity dependence)
Maximum number of levels of nesting #include files	40max
Maximum number of case names in one switch statement (with no nesting of switch statement)	No limit (Memory capacity dependence)
Total number of operators and operands that can be defined in #if and #elif	No limit (Memory capacity dependence)
Size of stack frame that can be secured per function(in bytes)	64K bytes max
Number of variables that can be defined in #pragma ADDRESS	No limit (Memory capacity dependence)
Maximum number of levels of nesting parentheses	No limit (Memory capacity dependence)
Number of initial values that can be defined when defining variables with initialization expressions	No limit (Memory capacity dependence)
Maximum number of levels of nesting modifier declarators	Depends on stack size of YACC
Maximum number of levels of nesting declarator parentheses	Depends on stack size of YACC
Maximum number of levels of nesting operator parentheses	Depends on stack size of YACC
Maximum number of valid characters per internal identifier or macro name	~~No limit (Memory capacity dependence)~~ 200 max
Maximum number of valid characters per external identifier	~~No limit (Memory capacity dependence)~~ 200 max
Maximum number of external identifiers per source file	No limit (Memory capacity dependence)

Table C.2 Translation Limits of Compiler (2/2)

Item	Specification
Maximum number of identifiers with block scope per block	No limit (Memory capacity dependence)
Maximum number of macros per source file	No limit (Memory capacity dependence)
Maximum number of parameters per function call and per function	No limit (Memory capacity dependence)
Maximum number of parameters or macro call parameters per macro	31max
Maximum number of characters in character string literals after concatenation	No limit (Memory capacity dependence)
Maximum size (in bytes) of object	No limit (Memory capacity dependence)
Maximum number of members per structure/union	No limit (Memory capacity dependence)
Maximum number of enumerator constants per enumerator	No limit (Memory capacity dependence)
Maximum number of levels of nesting of structures or unions per struct declaration list	No limit (Memory capacity dependence)
Maximum number of characters per character string	Depends on operating system
Maximum number of lines per file	No limit (Memory capacity dependence)
Maximum length of an identifier	200 characters

Appendix D C/C++ Language Specification Rules

This appendix describes the internal structure and mapping of data processed by NC30, the extended rules for signs in operations, etc., and the rules for calling functions and the values returned by functions.

D.1 Language Specifications

a. Keywords

This compiler interprets the following as keywords.

Keywords common to both C/C++ programs:

_asm	_far	_near	asm	auto
break	case	char	const	continue
default	do	double	else	enum
extern	far	float	for	goto
if	inline	int	long	near
register	return	short	signed	sizeof
static	struct	switch	typedef	union
unsigned	void	volatile	while	_inline

Keywords for C programs only:

_Bool	restrict	_ext4mptr

Keywords for C++ programs only:

bool	catch	class	const_cast	delete
dynamic_cast	explicit	false	friend	mutable
namespace	new	operator	private	protected
public	reinterpret_cast	static_cast	template	this
throw	true	try	typeid	typename
using	virtual	wchar_t	and	and_eq
bitand	bitor	compl	not	not_eq
or	or_eq	xor	xor_eq	-

In C++ programs, inline is handled as a keyword. When compiled as a C++ program, the compile option -fnot_reserve_inline has no effect.

b. Integer constants

Integer constants can be specified using octal, hexadecimal, and binary numbers, in addition to decimal numbers. The forms of respective numerical representations are listed in Table D.1

Table D.1 Rules for Writing Integer Constants

Representation	Rules	Composition	Example
Decimal	Begin with other than zero (0)	0123456789	15
Octal	Begin with zero (0)	01234567	017
Hexadecimal	Begin with 0X or 0x	0123456789abcdefABCDEF	0XF or 0xf
Binary	Begin with 0B or 0b	01	0B1 or 0b1

In binary representation, underscores '_' are ignored. They can be used as a visual delimiter.
Example: char port = 0b_0_111_1_011; /* Same value as 0b01111011 */

Types of integer constants are determined depending on the magnitude of values in the order given below.

- Octal, hexadecimal, and binary numbers
  signed int type → unsigned int type → signed long type → unsigned long type → singed long long type → unsigned long long type
- Decimal numbers (in C)
  signed int type → signed long type → signed long long type
  Decimal numbers (in C++)
  signed int type → signed long type → unsigned long type → signed long long type → unsigned long long type

Also, when numbers are suffixed with the letter U or u, L or l, or LL or ll , they are handled as described below.

(1) Unsigned constants
For unsigned constants, add the letter U or u after the constant value written. Types are determined depending on values in the order given below.
unsigned int type → unsigned long type → unsigned long long type

(2) Long-type constants
For long-type constants, add the letter L or l after the constant value written.Types are determined depending on values in the order given below.
- Octal, hexadecimal, and binary numbers
  signed long type → unsigned long type → singed long long type → unsigned long long type
- Decimal numbers (in C)
  signed long type → signed long long type
  Decimal numbers (in C++)
  signed long type → unsigned long type → signed long long type → unsigned long long type

(3) Long long-type constants
For long long-type constants, add the letters LL or ll after the constant value written.Types are determined depending on values in the order given below.
- Octal, hexadecimal, and binary numbers
  signed long long type → unsigned long long type
- Decimal numbers (in C)
  signed long long type
  Decimal numbers (in C++)
  signed long long type → unsigned long long type

D.2 Internal Representation of Data

D.2.1 Integral Type

Table D.2 shows the number of bytes used by integral type data

Table D.2 Data Size of Integral Type

Type	Existence of sign	Bit size	Range of values
_Bool	No	8	0, 1
char unsigned char	No	8	0 to 255
signed char	Yes	8	-128 to 127
int short signed int signed short	Yes	16	-32768 to 32767
unsigned int unsigned short	No	16	0 to 65535
long signed long	Yes	32	-2147483648 to 2147483647
unsigned long	No	32	0 to 4294967295
long long signed long long	Yes	64	-9223372036854775808 to 9223372036854775807
unsigned long long	No	64	0 to 18446744073709551615
bool	No	8	false, true
wchar_t	No	16	0 to 65535

- The _Bool type can not specify to sign.
- If a char type is specified with no sign, it is processed as an unsigned char type.
- If an int or short type is specified with no sign, it is processed as a signed int or signed short type.
- If a long type is specified with no sign, it is processed as a sign long type.
- If a long long type is specified with no sign, it is processed as a sign long long type.
- If the bit field members of a structure are specified with no sign, they are processed as unsigned.
- Can not specifies bit-fields of long long type.
- For _Bool and bool types, only bit 0 is used. The 7 high-order bits are indeterminate.

D.2.2 Floating Type

Table D.3 shows the number of bytes used by floating type data.

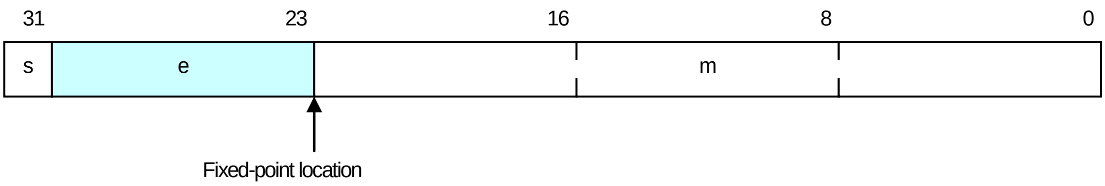
Table D.3 Data Size of Floating Type

Type	Existence of sign	Bit Size	Range of values
float	Yes	32	1.17549435e-38F to 3.40282347e+38F
double long double	Yes	64	2.2250738585072014e-308 to 1.7976931348623157e+308

When the compile option -fdouble_32(-fD32) is used, type double is assumed to be the same as type float. NC30's floating-point format conforms to the format of IEEE (Institute of Electrical and Electronics Engineers) standards. The following shows the single precision and double precision floating-point formats.

(1) Single-precision floating point data format

Figure D.1 shows the format for binary floating point (float) data.

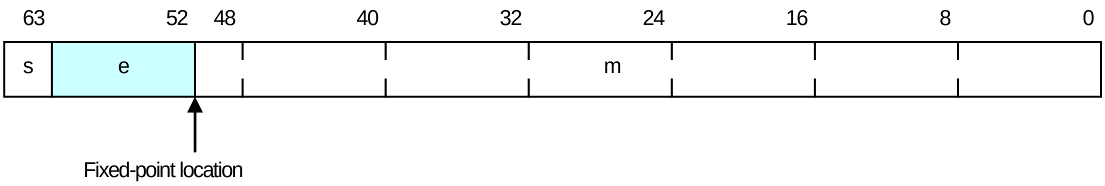


s: a sign part (1 bit)
e: a biased exponent part (8 bits)
m: a mantissa part (23 bits)

Figure D.1 Single-precision floating point data format

(2) Double-precision floating point data format

Figure D.2 shows the format for binary floating point (double and long double) data.



s: a sign part (1 bit)
e: a biased exponent part (11 bits)
m: a mantissa part (52 bits)

Figure D.2 Double-precision floating point data format

D.2.3 Enumerator Type

The enumerated type has the same internal representation as that of an unsigned int type in C or an int type in C++. Unless otherwise specified, integer numbers 0, 1, 2, and so on are assigned in the order in which members occur. The type of enumerators (enumerated members) is an int type, which is common to both C and C++.
Furthermore, by using the compile option -fchar_enumerator(-fCE), it is possible to let the enumerated type and the type of enumerators have the same internal representation as that of an unsigned char type.

D.2.4 Pointer Type

Table D.4 shows the number of bytes used by pointer type data.

Table D.4 Data Size of Pointer Types

Type	Existence of Sign	Bit Size	Range
near pointers	None	16	0 to 0xFFFF
far pointers	None	32	0 to 0xFFFFF

Note that only the least significant 20 bits of the 32 bits of far pointers are valid.

D.2.5 Array Types

Array types are mapped contiguously to an area equal to the product of the size of the elements (in bytes) and the number of elements. They are mapped to memory in the order in which the elements appear. Figure D.3 is an example of mapping.

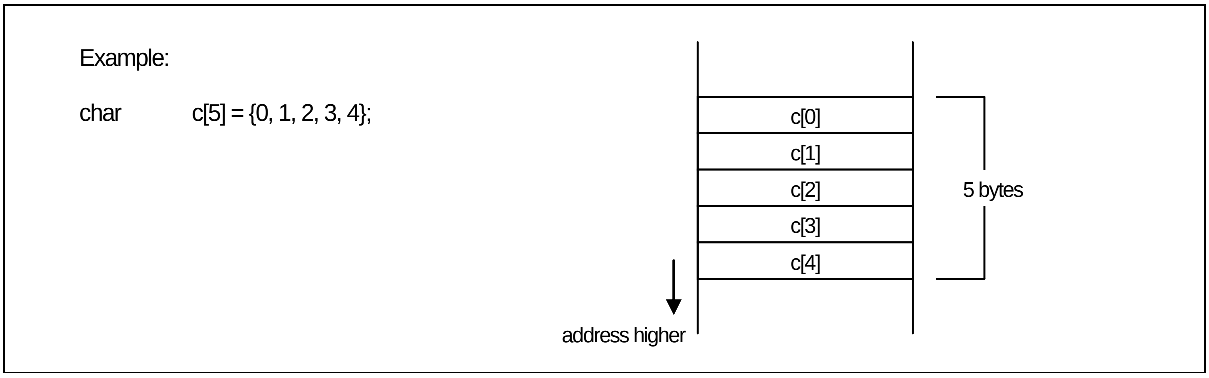


Figure D.3 Example of Placement of Array

D.2.6 Structure types

Structure types are mapped contiguously in the order of their member data. Figure D.4 is an example of mapping.

Example:

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

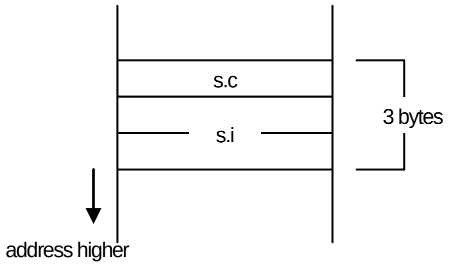


Figure D.4 Example of Placement of Structure (1)

Normally, there is no word alignment with structures. The members of structures are mapped contiguously. To use word alignment, use the #pragma STRUCT extended function. #pragma STRUCT adds a byte of padding if the total size of the members is odd. Figure D.5 is an example of mapping.

Example:

```
#pragma STRUCT   TAG   unpack

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

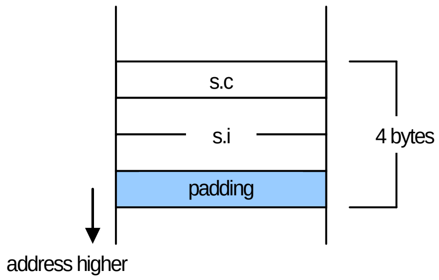


Figure D.5 Example of Placement of Structure (2)

D.2.7 Unions

Unions occupy an area equal to the maximum data size of their members. Figure D.6 is an example of mapping.

Example:

```
union TAG {
        char    c;
        int     i;
        long    lo;
} s;
```

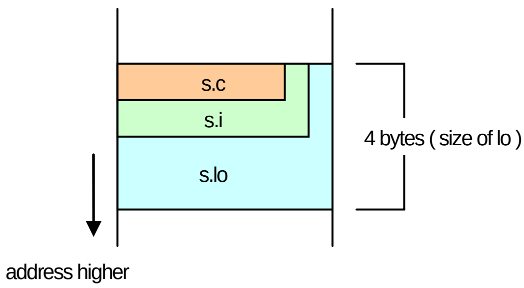


Figure D.6 Example of Placement of Union

D.2.8 Bitfield Types

Bitfield types are mapped from the least significant bit. Figure D.7 is an example of mapping.

Example:

```
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;
```

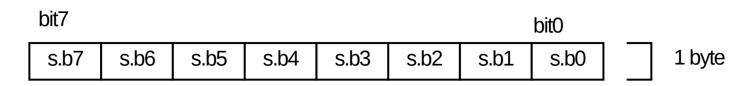


Figure D.7 Example of Placement of Bitfield (1)

If a bitfield member is of a different data type, it is mapped to the next address. Thus, members of the same data type are mapped contiguously from the lowest address to which that data type is mapped.

Example:

```
struct BTAG {
        char    b0 : 1;
        char    b1 : 1;
        char    b2 : 1;
        char    b3 : 1;
        char    b4 : 1;
        int     b56 :2;
        char    b7 : 1;
} s;
```

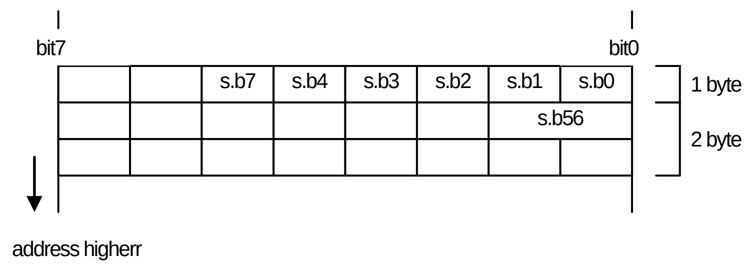


Figure D.8 Example of Placement of Bitfield (2)

Note :

(1) If no sign is specified, the default bitfield member type is unsigned.

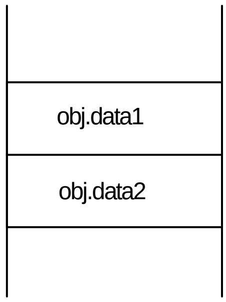
(2) Can not specifies bit-fields of long long type.

D.2.9 Class Types (C++)

For the base class and a class without virtual functions, the compiler allocates data members to memory according to the rules for structure data allocation.

Example:

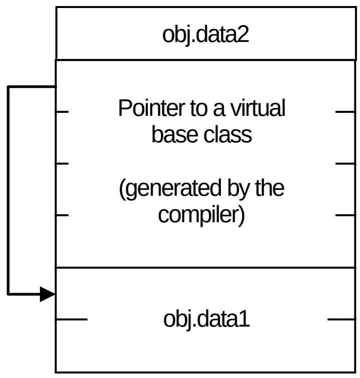
```
class A {
    char data1;
    char data2;
public:
    A();
    int get() { return data1; }
} obj;
```



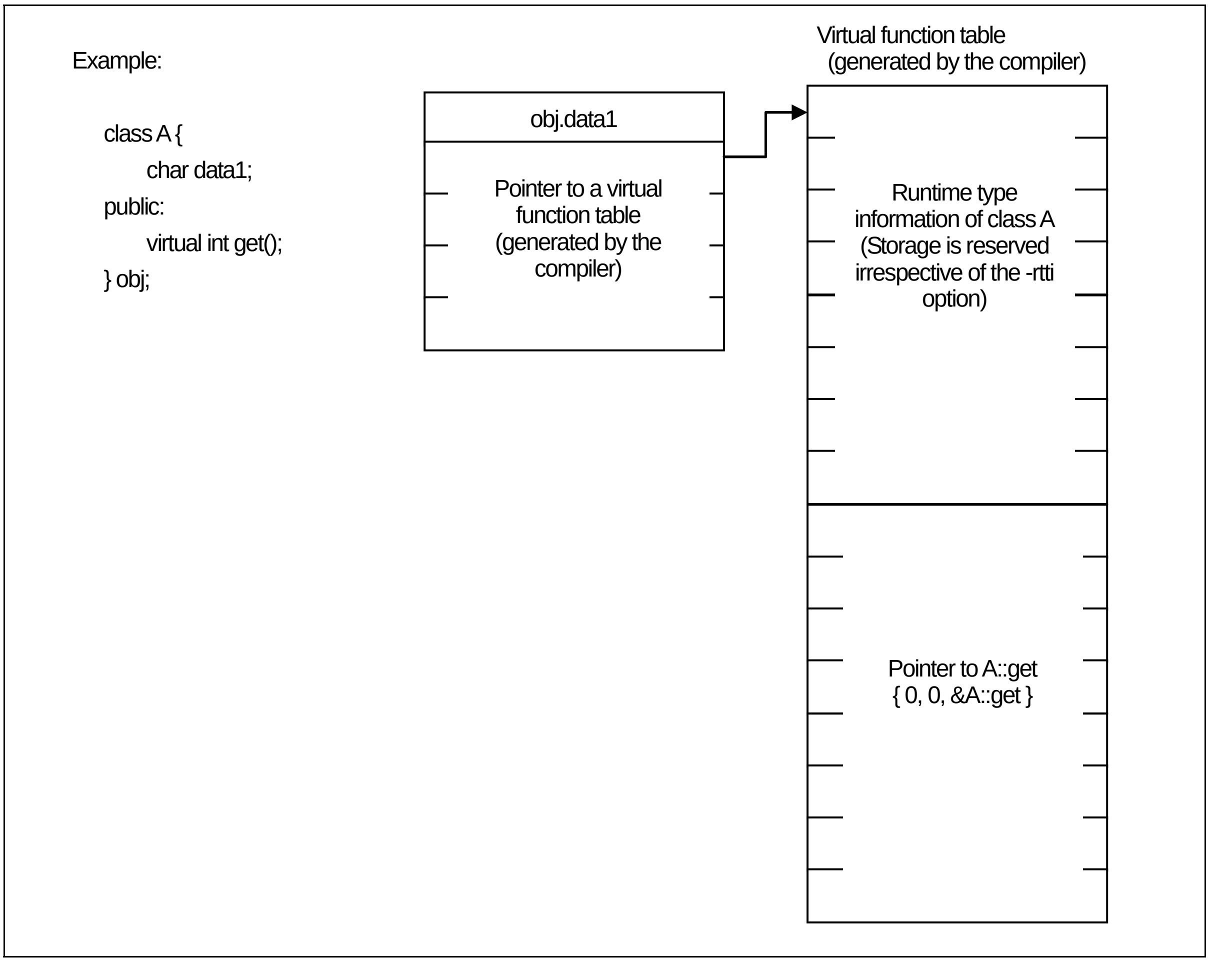
If the class has a virtual base class, the compiler assigns a pointer to the virtual base class.
The size of a pointer to a virtual base class is 2 bytes when the compile option -R8C is specified or 4 bytes when the compile option -R8C is not specified.

Example:

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



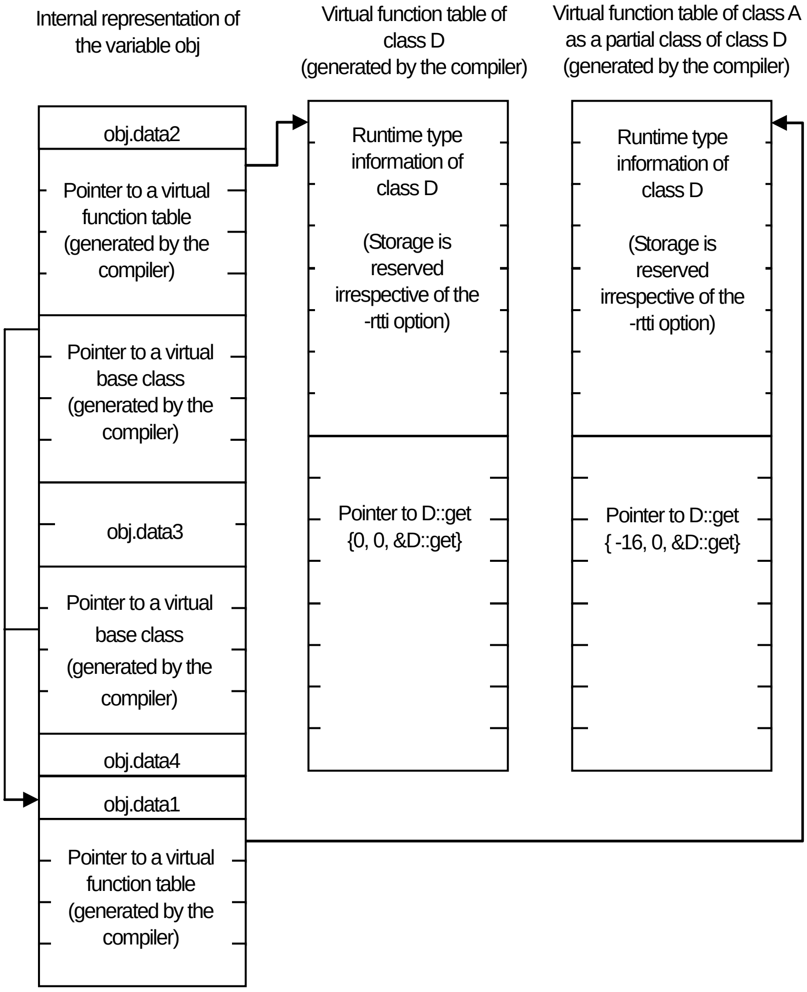
If the class has a virtual function, the compiler generates a virtual function table and assigns a pointer to the virtual function table.
The size of a pointer to a virtual function table is 2 bytes when the compile option -R8C is specified or 4 bytes when the compile option -R8C is not specified.


Example:
class A {
char data1;
public:
virtual int get();
} obj;
obj.data1
Pointer to a virtual function table (generated by the compiler)
Virtual function table (generated by the compiler)
Runtime type information of class A (Storage is reserved irrespective of the -rtti option)
Pointer to A::get
{ 0, 0, &A::get }

Shown below is an example where there are a virtual base class, base class, and a class with virtual function.

Example:

```
class A {
        char data1;
public:
        virtual short get();
};
class B : virtual A {
        char data2;
public:
        short get();
};
class C : virtual A {
        short data3;
};
class D : virtual A, B, C {
        char data4;
public:
        short get();
} obj;
```



For a empty class, the compiler allocates a 1-byte dummy area.

Example:

```
class A {
public:
        int fun();
} obj;
```

A empty class dummy area is allocated when the class size is 0. In cases when the base class or a derived class has data members, and for a class that has a virtual function, the compiler does not allocate a dummy area.

Example:

```
class A {
public:
        int fun();
};
class B : A {
public:
        char data1;
} obj;
```

obj.data1

Even for empty classes where the base class is a empty class, the dummy area consists of 1 byte.

Example:

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

obj. <dummy area>

D.2.10 Reference Type and Pointer-to-Member Type

Table D.5 shows the number of bytes that the data of reference type and pointer-to-member type uses.

Table D.5 Data Sizes of Reference Type and Pointer-to-Member Type

Type	Signed or not	Bit size	Representable numeric value
near reference	Not	16	-
far reference	Not	32	-
Pointer to data member	Not	16	0 to 0xFFFF
Pointer to function member	Not	64	-

D.3 Sign Extension Rules

Standard language specifications stipulate that the data of char, signed char, and unsigned char types should be sign-extended to int type when arithmetic operations, etc. are performed on data.
This compiler, by default, places emphasis on code efficiency and execution speed as it generates code, so that char, signed char, and unsigned char types are not extended to int type. Use of the compile option "-fansi" or "-fextend_to_int(-fETI)" nullifies this specification, allowing the compiler to perform sign extensions similar to the one in the standard C.
When writing an arithmetic operation that assigns the result of operation to char type as in Figure D.9 without using the compile option "-fansi" or "-fextend_to_int(-fETI)," be careful that the minimum and maximum values representable by char, signed char, or unsigned char do not overflow in the middle of operation.
When compiling the source in C++ mode, be aware that char, signed char, and unsigned char types are always type-converted to int type.
In the program here, the variable 'i' has 0x24 assigned to it when in C mode; when in C++ mode, the variable 'i' has 0x124 assigned to it.

Example:

```
{
    int i;
    char a = 0x8f;
    char b = 0x95;
    i = a + b;
 }
```

D.4 Function Call Rules

D.4.1 Rules of Return Value

When returning a return value from a function, the system uses a register to return that value for the integer, pointer, and floating-point types. Table D.6 shows rules on calls regarding return values.

Table D.6 Return Value-related Calling Rules

Type of Return Value	Rules
char type signed char type unsigned char type _Bool type bool type Enumerated type[1]	Returned in R0L register
signed short type unsigned short type signed int type unsigned int type near pointer type near reference type wchar_t type Enumerated type[2] Pointer-to-data members type	Returned in R0 register
float type double type[3] signed long type unsigned long type far pointer type far reference type	Least significant 16 bits returned by storing in R0 register. Most significant 16 bits returned by storing in R2 register.
double type[4] long double type	Returned in R3, R2, R1 and R0 registers, divided into 16-bit parts in that order beginning with the higher-order bits.
signed long long type unsigned long long type	Returned in R3, R1, R2 and R0 registers, divided into 16-bit parts in that order beginning with the higher-order bits.
Structure type Union type Class type Pointer-to-function members type	The far pointer indicating the area for storing a return value is saved to the stack immediately before making a call. The called function writes a return value to the saved area indicated by the far pointer before it returns.

[1] This applies only when one of -fchar_enumerator(-fCE), -OR_MAX(-ORM), or OS_MAX(-OSM) is specified.
[2] This applies only when none of -fchar_enumerator(-fCE), -OR_MAX(-ORM), or OS_MAX(-OSM) is specified.
[3] This applies only when one of -fdouble_32(-fD32), -OR_MAX(-ORM), or OS_MAX(-OSM) is specified.
[4] This applies only when none of -fdouble_32(-fD32), -OR_MAX(-ORM), or OS_MAX(-OSM) is specified.

D.4.2 Rules on Argument Transfer

NC30 uses registers or stack to pass arguments to a function.

(1) Passing arguments via register

When the conditions below are met, the system uses the corresponding "Registers Used" listed in Table D.7 to pass arguments.

- A prototype for the function has been declared[1] and parameter types have been made definite at the time of a function call.
- No variable parameters "…" are used in the prototype declaration.
- As parameter types for the function, the parameters and the types of parameters in Table D.7match.

Table D.7 Rules on Argument Transfer via Register (NC30)

Argument	First Argument	Registers Used
First argument	char type signed char type unsigned char type _Bool type bool type Enumerated type[2]	R1L register
	signed short type unsigned short type signed int type unsigned int type near pointer type near reference type wchar_t type Enumerated type[3] Pointer-to-data members type	R1 register
Second argument	signed short type unsigned short type signed int type unsigned int type near pointer type near reference type wchar_t type Enumerated type[4] Pointer-to-data members type	R2 register

[1] This compiler applies the method of passing arguments via register only when a function prototype is declared. If a function is written in K&R style, all arguments are passed via stack. Also, be aware that, for reasons of language specifications of C, if the manner of describing a function in the prototype declaration form and the K&R style of description coexist, arguments may not be passed to functions correctly.
Because of the above reason, we recommend that C source files be written in the prototype declaration form as a unified method of description.
[2] This applies only when one of -fchar_enumerator(-fCE), -OR_MAX(-ORM), or OS_MAX(-OSM) is specified.
[3] This applies only when none of -fchar_enumerator(-fCE), -OR_MAX(-ORM), or OS_MAX(-OSM) is specified.
[4] This applies only when none of -fchar_enumerator(-fCE), -OR_MAX(-ORM), or OS_MAX(-OSM) is specified.

(2) Passing arguments via stack

If there are some arguments that do not meet the pass-via-register condition, all of them are passed via stack.

D.4.3 Rules for Converting Functions into Assembly Language Symbols

The function names in which functions are defined in a C language source file are used as the start labels of functions in an assembler source file.
The beginning labels of functions in an assembler source file are a string consisting of the function name in a C source file and an "_" (underscore) or a $ (dollar mark) that is added at the top. The added strings and the condition under which they are added are shown in Table D.8.

Table D.8 Conditions Under Which Character Strings Are Added to Function

Added character string	Condition
$ (dollar)	Functions where any one of arguments is passed via register
_ (underscore)	Functions that do not belong to the above

Shown in Figure D.9 is a sample program where a function has register arguments and where a function has its arguments passed via only a stack.

```
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) Prototype declaration of the function func_proto
(2) Entity of the function func_proto (prototype for the function is declared)
(3) This is the body of function func_no_proto. (This is a description in K&R format, that is, an old format.)
(4) This is the body of function main.
(5) This calls function func_proto.
(6) This calls function func_no_proto.

Figure D.9 Sample Program for Calling a Function (sample.c)

As for the compilation result of the above sample program, a definition of the function func_proto (part (2)), a definition of the function func_no_proto (part (3)), and calls to the functions func_proto and func_no_proto (part (4)) are shown in Figure D.10, Figure D.11, and Figure D.12, respectively.
Use C linkage to reference the function names in a C++ program from an assembly program.
When a function is a C++ program is declared with C linkage (declared using extern "C"), the function can be referenced following the same rules as for C programs. However, the functions declared with C linkage cannot be overloaded.

```
;## #       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          ← (9)
;## #   REGISTER ARG    (        j)     size    2,    REGISTER R2          ← (8)
;## #       ARG Size(2)          Auto Size(4)          Context Size(5)

            .SECTION  program,CODE,ALIGN
            ._file    'sample.c'
            ._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) This passes the third argument k via stack.
(8) This passes the second argument j via register.
(9) This passes the first argument i via register.
(10) This is the start address of function func_proto.

Figure D.10 Compile Result of Sample Program (sample.c) (1)

In Figure D.10, since the function func_proto has its prototypes declared, the first and second arguments to it are passed via register. The third argument is passed via stack, because the pass-via-register rule does not apply to it.
Furthermore, as arguments to the function are passed via register, the symbol name for the start address of the function is "$func_proto," which is derived from the name "func_proto" written in the C source file by adding a $ (dollar mark) to it.

```
;## #       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)

            ._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) This passes all arguments via a stack.

(12) This is the start address of function func_no_proto.

Figure D.11 Compile Result of Sample Program (sample.c) (2)

In Figure D.11, since the function func_no_proto is written in K&R style, all arguments to it are passed via stack.

Furthermore, because there are no arguments to the function that are passed via register, the symbol name for the start address of the function is "_func_no_proto," which is derived from the name "func_no_proto" written in the C source file by adding an "_" (underscore) to it.

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

            ._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.b       #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.b       #06H,SP
            mov.w       R0,-2[FB]   ;  sum
            ._line      22
;## # C_SRC :           }
            exitd
E3:
            .align
            .END
```

Figure D.12 Compile Result of Sample Program (sample.c) (3)

Figure D.12 ,part[13]calls func_proto and part[14]calls func_no_proto.

D.4.4 Interface between Functions

For the program shown in Figure D.13, processes to build and free the stack frame are shown in Figure D.16 through Figure D.18. Note that Figure D.14 and Figure D.15 are the assembly language programs output as a result of the compilation of the program in Figure D.13.

```
int         func( int, int ,int);

void        main(void)
{
            int         i = 0x1234;             ← Argument to func
            int         j = 0x5678;             ← Argument to func
            int         k = 0x9abc;             ← Argument to func

            k = func( i, j ,k);
}

int         func( int x,int y,int z )
{
            int         sum;

            sum = 0;
            sum = x + y + z;
            return sum;                         ← Return value to main
}
```

Figure D.13   Example of C Language Sample Program

```
;## #      FUNCTION main
;## #      FRAME    AUTO     (      k)  size  2,   offset -6
;## #      FRAME    AUTO     (      j)  size  2,   offset -4
;## #      FRAME    AUTO     (      i)  size  2,   offset -2
;## #      ARG Size(0)       Auto Size(6)         Context Size(5)


           .SECTION program,CODE,ALIGN
           ._file   'sample.c'
           ._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.b    #02H,SP                               ← [10]
           mov.w    R0,-6 [FB] ;  k                       ← [11]
           ._line   10
;## # C_SRC :       }
           exitd
E1:
```

Figure D.14 Assembly language sample program (1)

```
;## #       FUNCTION func
;## #       FRAME    AUTO      (      sum) size   2,     offset -6
;## #       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(6)             Context Size(5)

            ._line      13
;## # C_SRC :          {
            .glb        $func
$func:
            enter       #06H                                              ← [7]
            mov.w       R1,-2[FB]   ;   x   x
            mov.w       R2,-4[FB]   ;   y   y
            ._line      16
;## # C_SRC :          sum = 0;
            mov.w       #0000H,-6[FB] ;   sum
            ._line      17
;## # 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,-6[FB]   ;   sum
            ._line      18
;## # C_SRC :                      return sum;
            mov.w       -6[FB],R0   ;   sum                               ← [8]
            exitd                                                         ← [9]
E2:
            .align
            .END
```

Figure D.15 Assembly language sample program (2)

The transitions of stack and register usage in processes (1) → (2) (processing on entry to the function "main") in Figure D.14, processes (3) → (4) → (5) → (6) → (7) (processing to call the function "func" and build the stack frame used by the function "func"), and processes (8) → (9) → (10) → (11) (processing to return from the function "func" to the function "main") in Figure D.15 are shown in Figure D.16, Figure D.17, and Figure D.18, respectively.

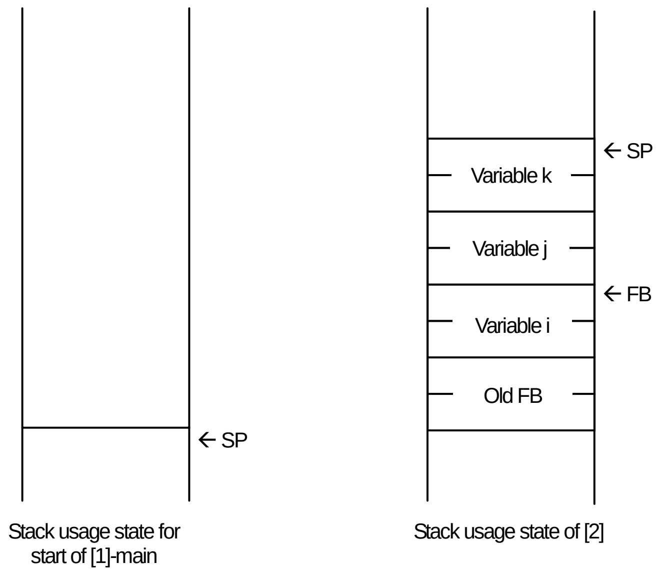


Figure D.16 Entry processing of function main

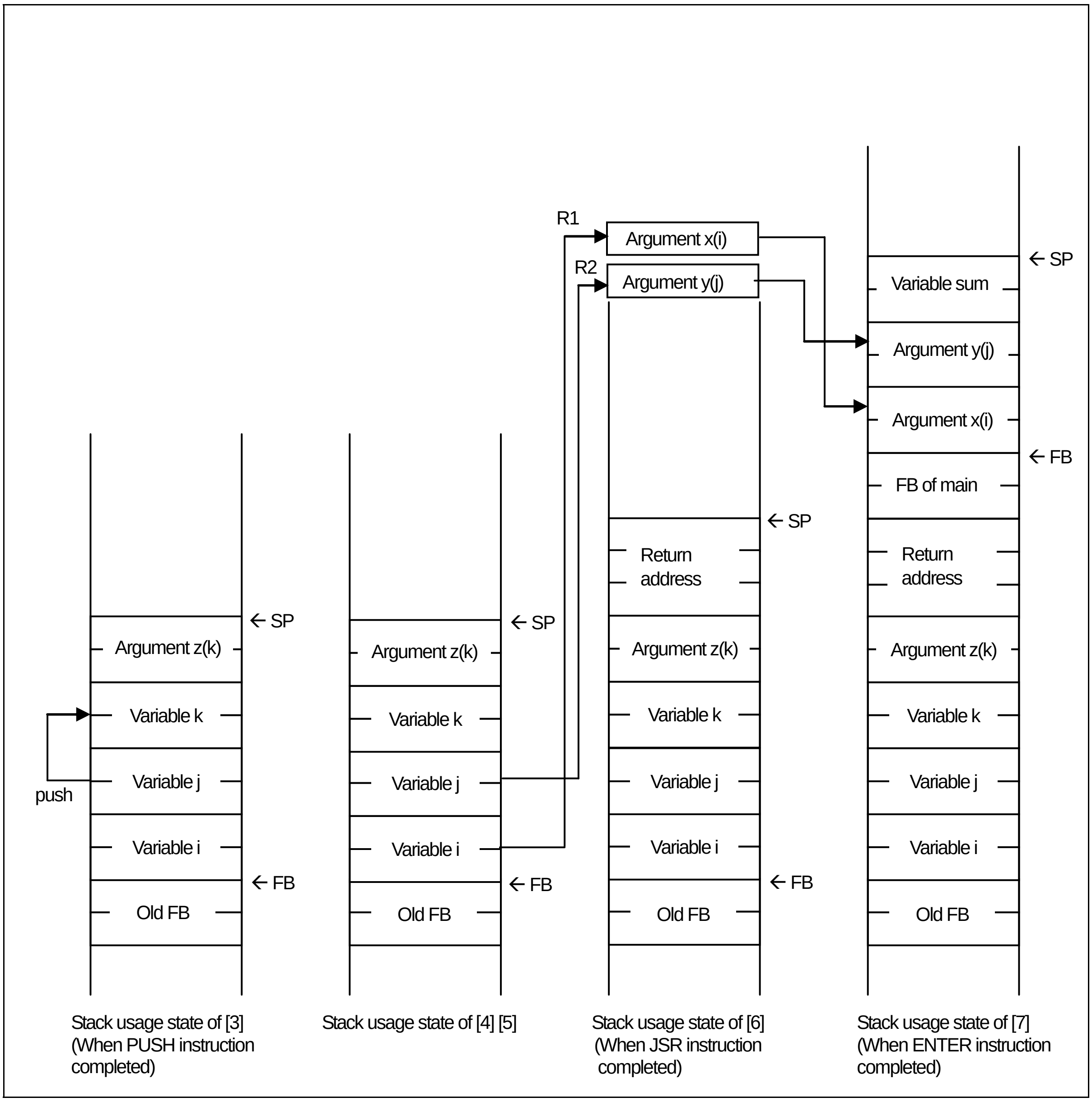


Figure D.17 Calling Function func and Entry Processing

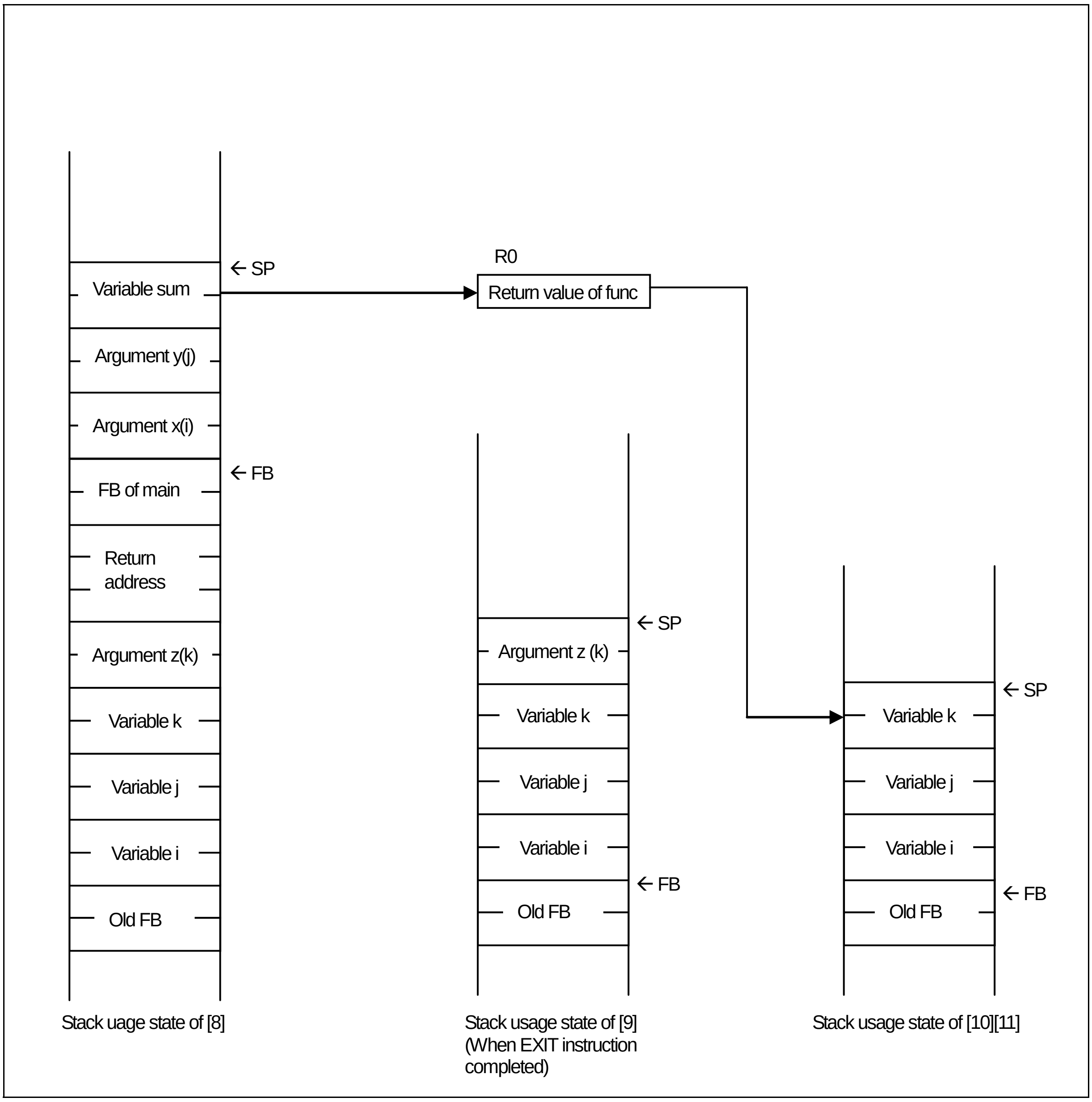


Figure D.18 Exit Processing of Function func

D.5 Securing auto Variable Area

Variables of storage class auto are placed in the stack of the micro processor. For a C language source file like the one shown in Figure D.19, if the areas where variables of storage class auto are valid do not overlap each other, the system allocates only one area which is then shared between multiple variables.

```
void        func(void)
{
            int         i, j, k;
            for( i=0 ; i<=0 ; i++ ){
                        process                                 scope of i
            }
                          :
                        (abbreviated)
                          :
            for( j=0xFF ; j<=0 ; j-- ){
                        process                                 scope of j
            }
                          :
                        (abbreviated)
                          :
            for( k=0 ; k<=0 ; k++ ){
                        process                                 scope of k
            }
}
```

Figure D.19 Example of C Program

In the example here, since the three auto variables 'i,' 'j,' and 'k' do not have their scopes overlapping one another, they share the same 2-byte area (offset position from the FB). The assembler source file generated by compiling Figure D.19 is shown in Figure D.20.

```
;### 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
                  :
             (remainder omitted)

* As shown by [1],[2], and [3],the three auto variables share the FB offset -2 area.
```

Figure D.20 Example of Assembler Source Program

D.6 Rules of Escaping of the Register

Rules for saving registers when calling a function are described below.

(1) The rules of Escaping of the register when call C function as follows:
- Register which use in called C function

(2) Register which should escaping in the entrance procedure of the called function.
- None

D.7 Preprocessor Specifications

D.7.1 Method for Loading an Include File

Syntax 1: #include△<file name>
Syntax 2: #include△"file name"

In syntax 1, a file in the directory specified with the startup option "-I" is included. If files cannot be found, the directory given below is searched.

- Standard directory set by the environment variable INC30

In syntax 2, a file is included from the directory that contains the files to be compiled. If files cannot be found, the directories given below are searched in order.

- Directory specified with the startup option "-I"
- Standard directory set by the environment variable INC30

D.7.2 Predefined Macros

The predefined macros are listed below.

__DATE__	Defines the date of compilation.
__FILE__	Defines the name of the source file.
__LINE__	Defines a line number in the source file.
__TIME__	Defines the time of compilation.
__STDC__	Defines 1 when the option -fansi is specified.
__RENESAS__	Always defines 1.
__RENESAS_VERSION__	Defines the version number of the compiler.
NC30	Always defines a space.
M16C	Always defines a space.
__R8C__	Defines a space when the option -R8C or -R8CE is specified.
__cplusplus	Defines 1 when a C++ program is compiled.

D.7.3 #assert

When a constant expression results in 0 (zero), the compiler outputs the following warning. It continues compiling as is.
sample.c(1) : C6696 (W) Assertion warning

D.8 Precautions to Take when Compiling a C++ Program

D.8.1 Precautions Regarding const-Qualified Variables

When compiling C++, the compiler does not necessarily locate the const-qualified variables in the rom section. The variables that accompany dynamic initialization are located in the bss section.

```
const int a = func();           // Locates the variable 'a' in the bss section.

const struct S {
        int a;
        S() {}
} b;                            // Locates the variable 'b' in the bss section.
```

D.8.2 Precautions about new/delete Operator Functions

The new/delete operator functions are called by a new/delete operator. In this compiler, the type of the return value of a new operator function and that of the first parameter of a delete operator are "void _far *".

```
struct S
{
    static void* operator new(size_t);
    static void   operator delete(void*);
};

void _far * alloc_int_S()
{
    void _far *(*pf)(size_t) = S::operator new;    // Since the type of the return value of a new operator function
                                                   // is implicitly a far pointer,
                                                   // if the RAM data pointer has the near attribute,
                                                   // a far qualification is required.
    return (*pf)(sizeof(int));
}

void dealloc_int_S(void _far * ptr)
{
    void (*pf)(void _far *) = S::operator delete;  // Since the first parameter of a delete operator function
                                                   // is implicitly a far pointer,
                                                   // if the RAM data pointer has the near attribute,
                                                   // a far qualification is required.
    (*pf)(ptr);
}
```

D.8.3 Precautions Regarding char Type

When compiled in C++, char type and unsigned char type are handled as separate types. Therefore, be aware that the source given below that could normally be compiled in a C program results in an error.

```
extern unsigned char port; // Declaration
                     char port; // Definition
```

For the sake of an increased portability of the source program, we recommend that, even in a C program, char type be used for types that represent characters, and singed char type or unsigned char type be used for types that represent 1-byte long integers.

D.8.4 Precautions Regarding a Description to Make near/far Definite in Multiple Declarations

When compiled as a C++ program, a description to make near/far attributes definite in multiple declarations—the one that was accepted when compiled as a C program—may result in an error.

```
extern int far fi;
int fi;                  // In C, the type of fi is interpreted as int far.
                         // In C++, if the RAM data location attribute is far, the type of fi is interpreted as int far;
                         //          if the RAM data location attribute is near, an error results.

extern int near ni;
int ni;                  // In C, the type of ni is interpreted as int near.
                         // In C++, if the RAM data location attribute is far, an error results;
                         //          if the RAM data location attribute is near, the type of ni is interpreted as int near.

extern int far * fpi;
int * fpi;               // In C, the type of fpi is interpreted as int far*.
                         // In C++, if the RAM data pointer attribute is far,
                         //          the type of fpi is interpreted as int far*;
                         //          if the RAM data pointer attribute is near, an error results.

extern int near * npi;
int * npi;               // In C, the type of npi is interpreted as int near*.
                         // In C++, if the RAM data pointer attribute is far, an error results;
                         //          if the RAM data pointer attribute is near,
                         //          the type of npi is interpreted as int near*.
```

D.8.5 Precautions Regarding Member Location Attributes near/far

The near/far declarations for variables with member location attributes, in a C program, are ignored. In a C++ program, an error results.

```
struct Tag {
        int near mem1;   // For C, near is ignored; for C++, an error results.
        int far    mem2;   // For C, far is ignored; for C++, an error results.
};
```

D.8.6 Precautions Regarding Inline Functions

In C++ compilation, the functions declared with the keyword "inline" and the member functions defined in a class definition are handled equally as are static functions and, therefore, not expanded in-line. To expand these functions in-line, use the compiler options -Ostatic_to_inline(-OSTI) and -Oforward_function_to_inline(-OFFTI). Note that if a function is defined in a global name space and the keyword "_inline" is used, the function is expanded in-line the same way as in C compilation.

D.8.7 Precautions Regarding the Location Attributes near/far of the Variables of Reference Type

Do not use the near/far qualifiers to specify the location attributes of the variables that have a reference type like the one shown below.

```
int   near   ni ;
int   near   & rni = ni ;              // OK
int   near   & near rni = ni ;         // NG
```

Appendix E C/C++ Library

E.1 Functionality of Each Standard Header File and Their Detailed Specifications

To use the standard library, it is necessary to include the header file in which its functions are declared. The functionality of each standard header file and their detailed specifications are described here.

E.1.1 Contents of Standard Header Files

This compiler comes with the standard header files listed in Table E.1

Table E.1 List of Standard Header Files

Header File Name	Contents
assert.h	Outputs the program's diagnostic information.
ctype.h	Declares character determination function as macro.
errno.h	Defines an error number.
float.h	Defines various limit values concerning the internal representation of floating points.
limits.h	Defines various limit values concerning the internal processing of compiler.
locale.h	Defines/declares macros and functions that manipulate program localization.
math.h	Declares arithmetic/logic functions for internal processing.
mathf.h	Declares arithmetic/logic functions for internal processing.(for float type)
setjmp.h	Defines the structures used in branch functions.
signal.h	Defines/declares necessary for processing asynchronous interrupts.
stdarg.h	Defines/declares the functions which have a variable number of real arguments.
stddef.h	Defines the macro names which are shared among standard include files.
stdio.h	(1) Defines the FILE structure. (2) Defines a stream name. (3) Declares prototypes for input/output functions
stdlib.h	Declares prototypes for memory management and termination functions
string.h	Declares prototypes for string and memory manipulating functions
time.h	Declares the functions necessary to indicate the current calendar time and defines the type.

E.1.2 Standard Header Files Reference

Following are detailed descriptions of the standard header files supplied with NC30. The header files are presented in alphabetical order.

The NC30 standard functions declared in the header files and the macros defining the limits of numerical expression of data types are described with the respective header files.

assert.h

Function: Defines the function macro "assert."

ctype.h

Function: Defines/declares string handling function.The following lists string handling functions.

Function	Contents
isalnum	Checks whether the character is an alphabet or numeral.
isalpha	Checks whether the character is an alphabet.
iscntrl	Checks whether the character is a control character.
isdigit	Checks whether the character is a numeral.
isgraph	Checks whether the character is printable (except a blank).
islower	Determines lowercase English letters
isprint	Checks whether the character is printable (including a blank).
ispunct	Checks whether the character is a punctuation character.
isspace	Checks whether the character is a blank, tab, or new line.
isupper	Checks whether the character is an upper-case letter.
isxdigit	Checks whether the character is a hexadecimal character.
tolower	Converts the character from an upper-case to a lower-case.
toupper	Converts the character from a lower-case to an upper-case.

errno.h

Function: Defines error number.

float.h

Function: Defines the limits of internal representation of floating point values. The following lists the macros that define the limits of floating point values.
In NC30, long double types are processed as double types. Therefore, the limits applying to double types also apply to long double types.

Macro name	Contents	Defined value
DBL_DIG	Maximum number of digits of double-type decimal precision	15
DBL_EPSILON	Minimum positive value where 1.0+DBL_EPSILON is found not to be 1.0	2.2204460492503131e-16
DBL_MANT_DIG	Maximum number of digits in the mantissa part when a double-type floating-point value is matched to the radix in its representation	53
DBL_MAX	Maximum value that a double-type variable can take on as value	1.7976931348623157e+308
DBL_MAX_10_EXP	Maximum value of the power of 10 that can be represented as a double-type floating-point numeric value	308
DBL_MAX_EXP	Maximum value of the power of the radix that can be represented as a double-type floating-point numeric value	1024
DBL_MIN	Minimum value that a double-type variable can take on as value	2.2250738585072014e-308
DBL_MIN_10_EXP	Minimum value of the power of 10 that can be represented as a double-type floating-point numeric value	-307
DBL_MIN_EXP	Minimum value of the power of the radix that can be represented as a double-type floating-point numeric value	-1021
FLT_DIG	Maximum number of digits of float-type decimal precision	6
FLT_EPSILON	Minimum positive value where 1.0+FLT_EPSILON is found not to be 1.0	1.19209290e-07F
FLT_MANT_DIG	Maximum number of digits in the mantissa part when a float-type floating-point value is matched to the radix in its representation	24
FLT_MAX	Maximum value that a float-type variable can take on as value	3.40282347e+38F
FLT_MAX_10_EXP	Maximum value of the power of 10 that can be represented as a float-type floating-point numeric value	38
FLT_MAX_EXP	Maximum value of the power of the radix that can be represented as a float-type floating-point numeric value	128
FLT_MIN	Minimum value that a float-type variable can take on as value	1.17549435e-38F
FLT_MIN_10_EXP	Minimum value of the power of 10 that can be represented as a float-type floating-point numeric value	-37
FLT_MIN_EXP	Maximum value of the power of the radix that can be represented as a float-type floating-point numeric value	-125
FLT_RADIX	Radix of exponent in floating-point representation	2
FLT_ROUNDS	Method of rounding off a floating-point number	1(Rounded to the nearest whole number)

Remarks:

- To use the compiler option -fdouble_32(-fD32), -OR_MAX(-ORM), or -OS_MAX(-OSM), define the same value for the DBL_XXX macros as in a FLT_XXX macro definition.
- The macros LDBL_XXX of long double type also are defined. Their definitions are the same as for DBL_XXX, except that the floating-point constants are suffixed by L.

limits.h

Function: Defines the limitations applying to the internal processing of the compiler. The following lists the macros that define these limits.

Macro name	Contents	Defined value
MB_LEN_MAX	Maximum value of the number of multibyte character- type bytes	1
CHAR_BIT	Number of char-type bits	8
CHAR_MAX	Maximum value that a char-type variable can take on as value	255
CHAR_MIN	Minimum value that a char-type variable can take on as value	0
SCHAR_MAX	Maximum value that a signed char-type variable can take on as value	127
SCHAR_MIN	Minimum value that a signed char-type variable can take on as value	-128
INT_MAX	Maximum value that a int-type variable can take on as valueMaximum value that a int-type variable can take on as value	32767
INT_MIN	Minimum value that a int-type variable can take on as value	-32768
SHRT_MAX	Maximum value that a short int-type variable can take on as value	32767
SHRT_MIN	Minimum value that a short int-type variable can take on as value	-32768
LONG_MAX	Maximum value that a long-type variable can take on as value	2147483647
LONG_MIN	Minimum value that a long-type variable can take on as value	-2147483648
LLONG_MAX	Maximum value that a signed long long-type variable can take on as value	9223372036854775807
LLONG_MIN	Minimum value that a signed long longtype variable can take on as value	-9223372036854775808
UCHAR_MAX	Maximum value that an unsigned char-type variable can take on as value	255
UINT_MAX	Maximum value that an unsigned int-type variable can take on as value	65535
USHRT_MAX	Maximum value that an unsigned short int-type variable can take on as value	65535
ULONG_MAX	Maximum value that an unsigned long int-type variable can take on as value	4294967295
ULLONG_MAX	Maximum value that an unsigned long long inttype variable can take on as value	18446744073709551615

locale.h

Function: Define/declares a macro function that handles the localization of a program.
The functions that have their prototypes declared are listed below.

Function	Contents
localeconv	Initializes struct lconv.
setlocale	Sets and searches the locale information of a program.

math.h (mathf.h)

Function: Declares prototype for a mathematic function.
The functions that have their prototypes declared are listed below.

Function	Contents
acos	Calculates arc cosine.
asin	Calculates arc sine.
atan	Calculates arc tangent.
atan2	Calculates arc tangent.
ceil	Calculates an integer carry value.
cos	Calculates cosine.
cosh	Calculates hyperbolic cosine.
exp	Calculates exponential function.
fabs	Calculates the absolute value of a double-precision floating-point number.
floor	Calculates an integer borrow value.
fmod	Calculates the remainder.
frexp	Divides floating-point number into mantissa and exponent parts.
ldexp	Calculates the power of a floating-point number.
log	Calculates natural logarithm.
log10	Calculates common logarithm.
modf	Calculates the division of a real number into the mantissa and exponent parts.
pow	Calculates the power of a number.
sin	Calculates sine.
sinh	Calculates hyperbolic sine.
sqrt	Calculates the square root of a numeric value.
tan	Calculates tangent.
tanh	Calculates hyperbolic tangent.

setjmp.h

Function: Declares prototype for a jump function and defines the structure used in that function.
The functions that have their prototypes declared are listed below.

Function	Contents
longjmp	Performs a global jump.
setjmp	Sets a stack environment for a global jump.

signal.h

Function: Defines/declares necessary for processing asynchronous interrupts.

stdarg.h

Function: Defines a macro and function used to process a variable-length parameter list.

stddef.h

Function: Defines the macro names which are shared among standard include files.

stdio.h

Function: Defines FILE structure/stream name and declares prototypes for input/output functions. The functions that have their prototypes declared are listed below.

Type	Function	Function
Initialize	clearerr	Initializes (clears) error status specifiers.
Input	fgetc	Inputs one character from the stream.
	getc	Inputs one character from the stream.
	getchar	Inputs one character from stdin.
	fgets	Inputs one line from the stream.
	gets	Inputs one line from stdin.
	fread	Inputs the specified items of data from the stream.
	scanf	Inputs characters with format from stdin.
	fscanf	Inputs characters with format from the stream.
	sscanf	Inputs data with format from a character string.
Output	fputc	Outputs one character to the stream.
	putc	Outputs one character to the stream.
	putchar	Outputs one character to stdout.
	fputs	Outputs one line to the stream.
	puts	Outputs one line to stdout.
	fwrite	Outputs the specified items of data to the stream.
	perror	Outputs an error message to stdout.
	printf	Outputs characters with format to stdout.
	fflush	Flushes the stream of an output buffer.
	fprintf	Outputs characters with format to the stream.
	sprintf	Writes text with format to a character string.
	vfprintf	Output to a stream with format.
	vprintf	Output to stdout with format.
	vsprintf	Output to a buffer with format.
Return	ungetc	Sends one character back to the input stream.
Determination	ferror	Checks input/output errors.
	feof	Checks EOF (End of File).

stdlib.h

Function: Declares prototypes for memory management and termination functions.
The functions that have their prototypes declared are listed below.

Function	Contents
abort	Terminates the execution of the program.
abs	Calculates the absolute value of an integer.
atof	Converts a character string into a double-type floating- point number.
atoi	Converts a character string into an int-type integer.
atol	Converts a character string into a long-type integer.
bsearch	Performs binary search in an array.
calloc	Allocates a memory area and initializes it to zero (0).
div	Divides an int-type integer and calculates the remainder.
free	Frees the allocated memory area.
labs	Calculates the absolute value of a long-type integer.
ldiv	Divides a long-type integer and calculates the remainder.
malloc	Allocates a memory area.
mblen	Calculates the length of a multibyte character string.
mbstowcs	Converts a multibyte character string into a wide character string.
mbtowc	Converts a multibyte character into a wide character.
qsort	Sorts elements in an array.
realloc	Changes the size of an allocated memory area.
strtod	Converts a character string into a double-type integer.
strtol	Converts a character string into a long-type integer.
strtoul	Converts a character string into an unsigned long-type integer.
wcstombs	Converts a wide character string into a multibyte character string.
wctomb	Converts a wide character into a multibyte character.

string.h

Function: Declares prototypes for string and memory manipulating functions.
The functions that have their prototypes declared are listed below.

Type	Type	Contents
Copy	strcpy	Copies a character string.
	strncpy	Copies a character string ('n' characters).
Concatenate	strcat	Concatenates character strings.
	strncat	Concatenates character strings ('n' characters).
Compare	strcmp	Compares character strings .
	strcoll	Compares character strings (using locale information).
	stricmp	Compares character strings. (All alphabets are handled as upper-case letters.)
	strncmp	Compares character strings ('n' characters).
	strnicmp	Compares character strings ('n' characters). (All alphabets are handled as upper-case letters.)
Search	strchr	Searches the specified character beginning with the top of the character string.
	strcspn	Calculates the length (number) of unspecified characters that are not found in the other character string.
	strpbrk	Searches the specified character in a character string from the other character string.
	strrchr	Searches the specified character from the end of a character string.
	strspn	Calculates the length (number) of specified characters that are found in the other character string.
	strstr	Searches the specified character from a character string.
	strtok	Divides some character string from a character string into tokens.
Length	strlen	Calculates the number of characters in a character string.
Convert	strerror	Converts an error number into a character string.
	strxfrm	Converts a character string (using locale information).
Initialize	bzero	Initializes a memory area (by clearing it to zero).
Copy	bcopy	Copies characters from a memory area to another.
	memcpy	Copies characters ('n' bytes) from a memory area to another.
	memset	Set a memory area by filling with characters.
Compare	memcmp	Compares memory areas ('n' bytes).
	memicmp	Compares memory areas (with alphabets handled as uppercase letters).
Search	memchr	Searches a character from a memory area.

time.h

Function: Declares the functions necessary to indicate the current calendar time and defines the type.

E.2 Standard Function Reference

Describes the features and detailed specifications of the standard function library of the compiler.

E.2.1 Overview of Standard Library

This compiler comes with the standard function library. These library functions are classified by functionality into the following kinds.

(1) String Handling Functions
Functions to copy and compare character strings, etc.

(2) Character Handling Functions
Functions to judge letters and decimal characters, etc., and to covert uppercase to lowercase and vice-versa.

(3) I/O Functions
Functions to input and output characters and character strings. These include functions for formatted I/O and character string manipulation.

(4) Memory Management Functions
Functions for dynamically securing and releasing memory areas.

(5) Memory Manipulation Functions
Functions to copy, set, and compare memory areas.

(6) Execution Control Functions
Functions to execute and terminate programs, and for jumping from the currently executing function to another function.

(7) Mathematical Functions
These functions require time.
- Therefore, pay attention to the use of the watchdog timer.

(8) Integer Arithmetic Functions
Functions for performing calculations on integer values.

(9) Character String Value Convert Functions
Functions for converting character strings to numerical values.

(10) Multi-byte Character and Multi-byte Character String Manipulate Functions
Functions for processing multi-byte characters and multi-byte character strings.

(11) Locale Functions
Locale-related functions.

E.2.2 List of Standard Library Functions by Function

a. String Handling Functions

The following lists String Handling Functions.

Table E.2 String Handling Functions

Type	Function	Contents	Reentrant
Copy	strcpy	Copies a character string.	○
	strncpy	Copies a character string ('n' characters).	○
Concatenate	strcat	Concatenates character strings.	○
	strncat	Concatenates character strings ('n' characters).	○
Compare	strcmp	Compares character strings .	○
	strcoll	Compares character strings (using locale information).	○
	stricmp	Compares character strings. (All alphabets are handled as upper-case letters.)	○
	strncmp	Compares character strings ('n' characters).	○
	strnicmp	Compares character strings ('n' characters). (All alphabets are handled as upper-case letters.)	○
Search	strchr	Searches the specified character beginning with the top of the character string.	○
	strcspn	Calculates the length (number) of unspecified characters that are not found in the other character string.	○
	strpbrk	Searches the specified character in a character string from the other character string.	○
	strrchr	Searches the specified character from the end of a character string.	○
	strspn	Calculates the length (number) of specified characters that are found in the other character string.	○
	strstr	Searches the specified character from a character string.	○
	strtok	Divides some character string from a character string into tokens.	×
Length	strlen	Calculates the number of characters in a character string.	○
Convert	strerror	Converts an error number into a character string.	×
	strxfrm	Copies a string (copies 'n' characters, locale information used)	○

b. Character Handling Functions

The following lists character handling functions.

Table E.3 Character Handling Functions

Function	Contents	Reentrant
isalnum	Checks whether the character is an alphabet or numeral.	○
isalpha	Checks whether the character is an alphabet.	○
iscntrl	Checks whether the character is a control character.	○
isdigit	Checks whether the character is a numeral.	○
isgraph	Checks whether the character is printable (except a blank).	○
islower	Determines lowercase English letters	○
isprint	Checks whether the character is printable (including a blank).	○
ispunct	Checks whether the character is a punctuation character.	○
isspace	Checks whether the character is a blank, tab, or new line.	○
isupper	Checks whether the character is an upper-case letter.	○
isxdigit	Checks whether the character is a hexadecimal character.	○
tolower	Converts the character from an upper-case to a lowercase.	○
toupper	Converts the character from a lower-case to an uppercase.	○

c. Input/Output Functions

The following lists Input/Output functions.

Table E.4 Input/Output Functions

Type	Function	Contents	Reentrant
Initialize	clearerror	Initializes (clears) error status specifiers.	×
Initialize	fgetc	Inputs one character from the stream.	×
	getc	Inputs one character from the stream.	×
	getchar	Inputs one character from stdin.	×
	fgets	Inputs one line from the stream.	×
	gets	Inputs one line from stdin.	×
	fread	Inputs the specified items of data from the stream.	×
	scanf	Inputs characters with format from stdin.	×
	fscanf	Inputs characters with format from the stream.	×
	sscanf	Inputs data with format from a character string.	×
Output	fputc	Outputs one character to the stream.	×
	putc	Outputs one character to the stream.	×
	putchar	Outputs one character to stdout.	×
	fputs	Outputs one line to the stream.	×
	puts	Outputs one line to stdout.	×
	fwrite	Outputs the specified items of data to the stream.	×
	perror	Outputs an error message to stdout.	×
	printf	Outputs characters with format to stdout.	×
	fflush	Flushes the stream of an output buffer.	×
	fprintf	Outputs characters with format to the stream.	×
	sprintf	Writes text with format to a character string.	×
	vfprintf	Output to a stream with format.	×
	vprintf	Output to stdout with format.	×
	vsprintf	Output to a buffer with format.	×
Return	ungetc	Sends one character back to the input stream.	×
Determination	ferror	Checks input/output errors.	×
	feof	Checks EOF (End of File).	×

d. Memory Management Functions

The following lists memory management functions.

Table E.5 Memory Management Functions

Function	Contents	Reentrant
calloc	Allocates a memory area and initializes it to zero (0).	×
free	Frees the allocated memory area.	×
malloc	Allocates a memory area.	×
realloc	Changes the size of an allocated memory area.	×

e. Memory Handling Functions

The following lists memory handling functions.

Table E.6 Memory Handling Functions

Type	Function	Contents	Reentrant
Initialize	bzero	Initializes a memory area (by clearing it to zero).	○
Copy	bcopy	Copies characters from a memory area to another.	○
	memcpy	Copies characters ('n' bytes) from a memory area to another.	○
	memset	Set a memory area by filling with characters.	○
Compare	memcmp	Compares memory areas ('n' bytes).	○
	memicmp	Compares memory areas (with alphabets handled as upper-case letters).	○
Move	memmove	Moves the area of a character string.	○
Search	memchr	Searches a character from a memory area.	○

f. Execution Control Functions

The following lists execution control functions.

Table E.7 Execution Control Functions

Function	Contents	Reentrant
abort	Terminates the execution of the program.	○
longjmp	Performs a global jump.	○
setjmp	Sets a stack environment for a global jump.	○

g. Mathematical Functions

The following lists mathematical functions.

Table E.8 Mathematical Functions

Function	Contents	Reentrant
acos	Calculates arc cosine.	×
asin	Calculates arc sine.	×
atan	Calculates arc tangent.	○
atan2	Calculates arc tangent.	×
ceil	Calculates an integer carry value.	○
cos	Calculates cosine.	○
cosh	Calculates hyperbolic cosine.	○
exp	Calculates exponential function.	○
fabs	Calculates the absolute value of a double-precision floating- point number.	○
floor	Calculates an integer borrow value.	○
fmod	Calculates the remainder.	○
frexp	Divides floating-point number into mantissa and exponent parts.	○
labs	Calculates the absolute value of a long-type integer.	○
ldexp	Calculates the power of a floating-point number.	○
log	Calculates natural logarithm.	×
log10	Calculates common logarithm.	×
modf	Calculates the division of a real number into the mantissa and exponent parts.	○
pow	Calculates the power of a number.	×
sin	Calculates sine.	○
sinh	Calculates hyperbolic sine.	○
sqrt	Calculates the square root of a numeric value.	×
tan	Calculates tangent.	○
tanh	Calculates hyperbolic tangent.	○

h. Integer Arithmetic Functions

The following lists integer arithmetic functions.

Table E.9 Integer Arithmetic Functions

Function	Contents	Reentrant
abs	Calculates the absolute value of an integer.	○
bsearch	Performs binary search in an array.	○
div	Divides an int-type integer and calculates the remainder.	○
labs	Calculates the absolute value of a long-type integer.	○
ldiv	Divides a long-type integer and calculates the remainder.	○
qsort	Sorts elements in an array.	×
rand	Generates a pseudo-random number.	○
srand	Imparts seed to a pseudo-random number generating routine.	○

i. Character String Value Convert Functions

The following lists character string value convert functions.

Table E.10 Character String Value Convert Functions

Function	Contents	Reentrant
atof	Converts a character string into a double-type floatingpoint number.	○
atoi	Converts a character string into an int	○
atol	Converts a character string into a long	○
strtod	Converts a character string into a double	○
strtol	Converts a character string into a long	○
strtou	Converts a character string into an unsigned long-type integer.	○

j. Multi-byte Character and Multi-byte Character String Manipulate Functions

The following lists Multibyte Character and Multibyte Character string Manipulate Functions.

Table E.11 Multibyte Character and Multibyte Character String Manipulate Functions

Function	Contents	Reentrant
mblen	Calculates the length of a multibyte character string.	○
mbstowcs	Converts a multibyte character string into a wide character string.	○
mbtowc	Converts a multibyte character into a wide character.	○
wcstombs	Converts a wide character string into a multibyte character string.	○
wctomb	Converts a wide character into a multibyte character.	○

k. Localization Functions

The following lists localization functions.

Table E.12 Localization Functions

Function	Contents	Reentrant
localeconv	Initializes struct lconv.	○
setlocale	Sets and searches the locale information of a program.	○

E.2.3 Standard Function Reference

The following describes the detailed specifications of the standard functions provided in NC30. The functions are listed in alphabetical order.
Note that the standard header file (extension .h) shown under "Format" must be included when that function is used.

A

abort

Execution Control Functions

Function:	Terminates the execution of the program abnormally.
Format:	#include <stdlib.h>
	void abort(void);
Method:	function
Variable:	No argument used.
ReturnValue:	No value is returned.
Description:	Terminates the execution of the program abnormally.
Note:	Actually, the program loops in the abort function.

abs

Integer Arithmetic Functions

Function:	Calculates the absolute value of an integer.	
Format:	#include <stdlib.h>	
	int abs(n);	
Method:	function	
Variable:	int n;	Integer
ReturnValue:	Returns the absolute value of integer n (distance from 0).	

acos

Mathematical Functions

Function: Calculates arc cosine.

Format: #include <math.h>

double acos(x);

Method: function

Variable: double x; arbitrary real number

ReturnValue:
- Assumes an error and returns 0 if the value of given real number x is outside therange of -1.0 to 1.0.
- Otherwise, returns a value in the range from 0 to p radian.

asin

Mathematical Functions

Function: Calculates arc sine.

Format: #include <math.h>

double asin(x);

Method: function

Variable: double x; arbitrary real number

ReturnValue:
- Assumes an error and returns 0 if the value of given real number x is outside the range of -1.0 to 1.0.
- Otherwise, returns a value in the range from -p/2 to p/2 radian.

atan

Mathematical Functions

Function: Calculates arc tangent.

Format: #include <math.h>

double atan(x);

Method: function

Variable: double x; arbitrary real number

ReturnValue: Returns a value in the range from -π/2 to π/2 radian.

atan2

Mathematical Functions

Function:	Calculates arc tangent.	
Format:	#include <math.h>	
	double atan2(x , y);	
Method:	function	
Variable:	double x;	arbitrary real number
	double y;	arbitrary real number
ReturnValue:	Returns a value in the range from -π to π radian.	

atof

Character String Value Convert Functions

Function:	Converts a character string into a double-type floating- point number.	
Format:	#include <stdlib.h>	
	double atof(s);	
Method:	function	
Variable:	const char _far *s;	Pointer to the converted character string
ReturnValue:	A string converted into a double-precision floating number is returned.	

atoi

Character String Convert Functions

Function: Converts a character string into an int-type integer.

Format: #include <stdlib.h>

int atoi(s);

Method: function

Variable: const char _far *s; Pointer to the converted character string

ReturnValue: Returns the value derived by converting a character string into an int-type integer.

atol

Character String Convert Functions

Function: Converts a character string into a long-type integer.

Format: #include <stdlib.h>

long atol(s);

Method: function

Variable: const char _far *s; Pointer to the converted character string

ReturnValue: Returns the value derived by converting a character string into a long-type integer.

B

bcopy

Memory Handling Functions

Function:	Copies characters from a memory area to another.	
Format:	#include <string.h>	
	void bcopy(src, dtop, size);	
Method:	function	
Variable:	char _far *src;	Start address of the memory area to be copied from
	char _far *dtop;	Start address of the memory area to be copied to
	unsigned long size;	Number of bytes to be copied
ReturnValue:	Copies the number of bytes specified in size from the beginning of the area specified in src to the area specified in dtop.	

bsearch

Integer Arithmetic Functions

Function:	Searches an array for elements.	
Format:	#include <stdlib.h>	
	void *bsearch(key, base, nelem, size, cmp);	
Method:	function	
Variable:	const void _far *key;	Search key
	const void _far *base;	Start address of array
	size_t nelem;	Element number
	size_t size;	Element size
	int cmp();	Compare function
ReturnValue:	● Returns a pointer to an array element that equals the search key. ● Returns a NULL pointer if no elements matched.	
Note:	The specified item is searched from the array after it has been sorted in ascending order.	

bzero

Memory Handling Functions

Function:	Initializes a memory area (by clearing it to zero).	
Format:	#include <string.h> void bzero(top, size);	
Method:	function	
Variable:	char _far *top; unsigned long size;	Start address of the memory area to be cleared to zero Number of bytes to be cleared to zero
ReturnValue:	No value is returned.	
Description:	Initializes (to 0) the number of bytes specified in size from the starting address of the area specified in top.	

C

calloc

Memory Management Functions

Function:	Allocates a memory area and initializes it to zero (0).
Format:	#include <stdlib.h>
	void _far * calloc(n, size);
Method:	function
Variable:	size_t n; Number of elements size_t size; Value indicating the element size in bytes
ReturnValue:	Returns NULL if a memory area of the specified size could not be allocated.
Description:	● After allocating the specified memory, it is cleared to zero. ● The size of the memory area is the product of the two parameters.
Rule:	The rules for securing memory are the same as for malloc.

ceil

Mathematical Functions

Function:	Calculates an integer carry value.
Format:	#include <math.h>
	double ceil(x);
Method:	function
Argument:	double x; arbitrary real number
ReturnValue:	Returns the minimum integer value from among integers larger than given real number x.

clearerr

Input/Output Functions

Function: Initializes (clears) error status specifiers.

Format: #include <stdio.h>

void clearerr(stream);

Method: function

Argument: FILE _far *stream; Pointer of stream

ReturnValue: No value is returned.

Description: Resets the error designator and end of file designator to their normal values.

cos

Mathematical Functions

Function: Calculates cosine.

Format: #include <math.h>

double cos(x);

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the cosine of given real number x handled in units of radian.

cosh

Mathematical Functions

Function: Calculates hyperbolic cosine.

Format: #include <math.h>

double cosh(x);

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the hyperbolic cosine of given real number x.

D

div

Integer Arithmetic Functions

Function:	Divides an int-type integer and calculates the remainder.	
Format:	#include <stdlib.h>	
	div_t div(number, denom);	
Method:	function	
Argument:	int number;	Dividend
	int denom;	Divisor
ReturnValue:	Returns the quotient derived by dividing "number" by "denom" and the remainder of the division.	

Description:

- Returns the quotient derived by dividing "number" by "denom" and the remainder of the division in structure div_t.
- div_t is defined in stdlib.h. This structure consists of members int quot and int rem.

E

exp

Mathematical Functions

Function: Calculates exponential function.

Format: #include <math.h>

double exp(x);

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the calculation result of an exponential function of given real number x.

F

fabs

Mathematical Functions

Function:	Calculates the absolute value of a double-precision floating-point number.
Format:	#include <math.h>
	double fabs(x);
Method:	function
Argument:	double x; arbitrary real number
ReturnValue:	Returns the absolute value of a double-precision floating-point number.

feof

Input/Output Functions

Function: Checks EOF (End of File).

Format: #include <stdio.h>

int feof(stream);

Method: macro

Argument: FILE _far *stream; Pointer of stream

ReturnValue:

- Returns "true" (other than 0) if the stream is EOF.
- Otherwise, returns NULL (0).

Description:

- Determines if the stream has been read to the EOF.
- Interprets code 0x1A as the end code and ignores any subsequent data.

ferror

Input/Output Functions

Function: Checks input/output errors.

Format: #include <stdio.h>

int ferror(stream);

Method: macro

Argument: FILE _far *stream; Pointer of stream

ReturnValue:
- Returns "true" (other than 0) if the stream is in error.
- Otherwise, returns NULL (0).

Description:
- Determines errors in the stream.
- Interprets code 0x1A as the end code and ignores any subsequent data.

fflush

Input/Output Functions

Function: Flushes the stream of an output buffer.

Format: #include <stdio.h>

int fflush(stream);

Method: function

Argument: FILE _far *stream; Pointer of stream

ReturnValue: Always returns 0.

fgetc

Input/Output Functions

Function: Reads one character from the stream.

Format: #include <stdio.h>

int fgetc(stream);

Method: function

Argument: FILE _far *stream; Pointer of stream

ReturnValue:
- Returns the one input character.
- Returns EOF if an error or the end of the stream is encountered.

Description:
- Reads one character from the stream.
- Interprets code 0x1A as the end code and ignores any subsequent data.

fgets

Input/Output Functions

Function: Reads one line from the stream.

Format: #include <stdio.h>

char _far * fgets(buffer, n, stream);

Method: function

Argument:

char _far *buffer;	Pointer of the location to be stored in
int n;	Maximum number of characters
FILE _far *stream;	Pointer of stream

ReturnValue:

- Returns the pointer of the location to be stored (the same pointer as given by the argument) if normally input.
- Returns the NULL pointer if an error or the end of the stream is encountered.

Description:

- Reads character string from the specified stream and stores it in the buffer
- Input ends at the input of any of the following:
  (1) new line character ('\n')
  (2) n-1 characters
  (3) end of stream
- A null character ('\0') is appended to the end of the input character string.
- The new line character ('\n') is stored as-is.
- Interprets code 0x1A as the end code and ignores any subsequent data.

floor

Mathematical Functions

Function: Calculates an integer borrow value.

Format: #include <math.h>

double floor(x);

Method: function

Argument: double x; arbitrary real number

ReturnValue: The real value is truncated to form an integer, which is returned as a double type.

fmod

Mathematical Functions

Function: Calculates the remainder.

Format: #include <math.h>

double fmod(x ,y);

Method: function

Argument:

double x;	dividend
double y;	divisor

ReturnValue: Returns a remainder that derives when dividend x is divided by divisor y.

fprintf

Input/Output Functions

Function: Outputs characters with format to the stream.

Format: #include <stdio.h>

int fprintf(stream, format, argument...);

Method: function

Argument:

FILE _far *stream;	Pointer of stream
const char _far *format;	Pointer of the format specifying character string

ReturnValue:

- Returns the number of characters output.
- Returns EOF if a hardware error occurs.

Description:

- Argument is converted to a character string according to format and output to the stream.
- Format is specified in the same way as in printf.

fputc

Input/Output Functions

Function: Outputs one character to the stream.

Format: #include <stdio.h>

int fputc(c, stream);

Method: function

Argument:

int c;	Character to be output
FILE _far *stream;	Pointer of the stream

ReturnValue:

- Returns the output character if output normally.
- Returns EOF if an error occurs.

Description: Outputs one character to the stream.

fputs

Input/Output Functions

Function:	Outputs one line to the stream.	
Format:	#include <stdio.h>	
	int fputs (str, stream);	
Method:	function	
Argument:	const char _far *str;	Pointer of the character string to be output
	FILE _far *stream;	Pointer of the stream
ReturnValue:	● Returns 0 if output normally. ● Returns any value other than 0 (EOF) if an error occurs.	
Description:	Outputs one line to the stream.	

fread

Input/Output Functions

Function:	Reads fixed-length data from the stream	
Format:	#include <stdio.h>	
	size_t fread(buffer, size, count, stream);	
Method:	function	
Argument:	void _far *buffer;	Pointer of the location to be stored in
	size_t size;	Number of bytes in one data item
	size_t count;	Maximum number of data items
	FILE _far *stream;	Pointer of stream
ReturnValue:	Returns the number of data items input.	
Description:	● Reads data of the size specified in size from the stream and stores it in the buffer. This is repeated by the number of times specified in count. ● If the end of the stream is encountered before the data specified in count has been input, this function returns the number of data items read up to the end of the stream. ● Interprets code 0x1A as the end code and ignores any subsequent data.	

free

Memory Management Function

Function:	Frees the allocated memory area.	
Format:	#include <stdlib.h>	
	void free(cp);	
Method:	function	
Argument:	void _far *cp;	Pointer to the memory area to be freed
ReturnValue:	No value is returned.	

Description:

- Frees memory areas previously allocated with malloc or calloc.
- No processing is performed if you specify NULL in the parameter.

frexp

Mathematical Functions

Function:	Divides floating-point number into mantissa and exponent parts.	
Format:	#include <math.h>	
	double frexp(x,prexp);	
Method:	function	
Argument:	double x;	float-point number
	int _far *prexp;	Pointer to an area for storing a 2-based exponent
ReturnValue:	Returns the floating-point number x mantissa part.	

fscanf

Input/Output Function

Function: Reads characters with format from the stream.

Format: #include <stdio.h>

int fscanf(stream, format, argument...);

Method: function

Argument:

FILE _far *stream;	Pointer of stream
const char _far *format;	Pointer of the input character string

ReturnValue:

- Returns the number of data entries stored in each argument.
- Returns EOF if EOF is input from the stream as data.

Description:

- Converts the characters input from the stream as specified in format and stores them in the variables shown in the arguments.
- Argument must be a pointer to the respective variable.
- Interprets code 0x1A as the end code and ignores any subsequent data.
- Format is specified in the same way as in scanf.

fwrite

Input/Output Functions

Function: Outputs the specified items of data to the stream.

Format: #include <stdio.h>

size_t fwrite(buffer, size, count, stream);

Method: function

Argument:

const void _far *buffer;	Pointer of the output data
size_t size;	Number of bytes in one data item
size_t count;	Maximum number of data items
FILE _far *stream;	Pointer of the stream

ReturnValue: Returns the number of data items output

Description:

- Outputs data with the size specified in size to the stream. Data is output by the number of times specified in count.
- If an error occurs before the amount of data specified in count has been input, this function returns the number of data items output to that point.

G

getc

Input/Output Functions

Function: Reads one character from the stream.

Format: #include <stdio.h>

int getc(stream);

Method: macro

Argument: FILE _far *stream; Pointer of stream

ReturnValue:
- Returns the one input character.
- Returns EOF if an error or the end of the stream is encountered.

Description:
- Reads one character from the stream.
- Interprets code 0x1A as the end code and ignores any subsequent data.

getchar

Input/Output Functions

Function: Reads one character from stdin.

Format: #include <stdio.h>

int getchar(void);

Method: macro

Argument: No argument used.

ReturnValue:
- Returns the one input character.
- Returns EOF if an error or the end of the file is encountered.

Description:
- Reads one character from stream(stdin).
- Interprets code 0x1A as the end code and ignores any subsequent data.

gets

Input/Output Functions

Function: Reads one line from stdin.

Format: #include <stdio.h>

char _far * gets(buffer);

Method: function

Argument: char _far *buffer; Pointer of the location to be stored in

ReturnValue:
- Returns the pointer of the location to be stored (the same pointer as given by the argument) if normally input.
- Returns the NULL pointer if an error or the end of the file is encountered.

Description:
- Reads character string from stdin and stores it in the buffer.
- The new line character ('\n') at the end of the line is replaced with the null character ('\0').
- Interprets code 0x1A as the end code and ignores any subsequent data.

I

isalnum

Character Handling Functions

Function: Checks whether the character is an alphabet or numeral(A - Z,a - z,0 - 9).

Format: #include <ctype.h>

int isalnum(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if an alphabet or numeral.
- Returns 0 if not an alphabet nor numeral.

Description: Determines the type of character in the parameter.

isalpha

Character Handling Functions

Function: Checks whether the character is an alphabet(A - Z,a - z).

Format: #include <ctype.h>

int isalpha(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if an alphabet.
- Returns 0 if not an alphabet.

Description: Determines the type of character in the parameter.

iscntrl

Character Handling Functions

Function: Checks whether the character is a control character(0x00 - 0x1f,0x7f).

Format: #include <ctype.h>

int iscntrl(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a numeral.
- Returns 0 if not a control character.

Description: Determines the type of character in the parameter.

isdigit

Character Handling Functions

Function: Checks whether the character is a numeral(0 - 9).

Format: #include <ctype.h>

int isdigit(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a numeral.
- Returns 0 if not a numeral.

Description: Determines the type of character in the parameter.

isgraph

Character Handling Functions

Function: Checks whether the character is printable (except a blank)(0x21 - 0x7e).

Format: #include <ctype.h>

int isgraph(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if printable.
- Returns 0 if not printable.

Description: Determines the type of character in the parameter.

islower

Character Handling Functions

Function: Checks whether the character is a lower-case letter(a - z).

Format: #include <ctype.h>

int islower(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a lower-case letter.
- Returns 0 if not a lower-case letter.

Description: Determines the type of character in the parameter.

isprint

Character Handling Functions

Function: Checks whether the character is printable (including a blank)(0x20 - 0x7e).

Format: #include <ctype.h>

int isprint(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if printable.
- Returns 0 if not printable.

Description: Determines the type of character in the parameter.

ispunct

Character Handling Functions

Function: Checks whether the character is a punctuation character.

Format: #include <ctype.h>

int ispunct(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a punctuation character.
- Returns 0 if not a punctuation character.

Description: Determines the type of character in the parameter.

isspace

Character Handling Functions

Function: Checks whether the character is a blank, tab, or new line.

Format: #include <ctype.h>

int isspace(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a blank, tab, or new line.
- Returns 0 if not a blank, tab, or new line.

Description: Determines the type of character in the parameter.

isupper

Character Handling Functions

Function: Checks whether the character is an upper-case letter(A - Z).

Format: #include <ctype.h>

int isupper(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if an upper-case letter.
- Returns 0 if not an upper-case letter.

Description: Determines the type of character in the parameter.

isxdigit

Character Handling Functions

Function: Checks whether the character is a hexadecimal character(0 - 9,A - F,a - f).

Format: #include <ctype.h>

int isxdigit(c);

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a hexadecimal character.
- Returns 0 if not a hexadecimal character.

Description: Determines the type of character in the parameter.

L

labs

Integer Arithmetic Functions

Function:	Calculates the absolute value of a long-type integer.	
Format:	#include <stdlib.h>	
	long labs(n);	
Method:	function	
Argument:	long n;	Long integer
ReturnValue:	Returns the absolute value of a long-type integer (distance from 0).	

ldexp

Localization Functions

Function:	Calculates the power of a floating-point number.	
Format:	#include <math.h>	
	double ldexp(x,exp);	
Method:	function	
Argument:	double x; int exp;	Float-point number Power of number
ReturnValue:	Returns x *(exp power of 2).	

ldiv

Integer Arithmetic Functions

Function: Divides a long-type integer and calculates the remainder.

Format: #include <stdlib.h>

ldiv_t ldiv(number, denom);

Method: function

Argument:

long number;	Dividend
long denom;	Divisor

ReturnValue: Returns the quotient derived by dividing "number" by "denom" and the remainder of the division.

Description:

- Returns the quotient derived by dividing "number" by "denom" and the remainder of the division in the structure ldiv_t.
- ldiv_t is defined in stdlib.h. This structure consists of members long quot and long rem.

localeconv

Localization Functions

Function: Initializes struct lconv.

Format: #include <locale.h>

struct lconv _far *localeconv(void);

Method: function

Argument: No argument used.

ReturnValue: Returns a pointer to the initialized struct lconv.

log

Mathematical Functions

Function: Calculates natural logarithm.

Format: #include <math.h>

double log(x);

Method: function

Argument:

double x;	arbitrary real number

ReturnValue: Returns the natural logarithm of given real number x.

Description: This is the reverse function of exp.

log10

Mathematical Functions

Function:	Calculates common logarithm.	
Format:	#include <math.h>	
	double log10(x);	
Method:	function	
Argument:	double x;	arbitrary real number
ReturnValue:	Returns the common logarithm of given real number	

longjmp

Execution Control Functions

Function:	Restores the environment when making a function call	
Format:	#include <setjmp.h>	
	void longjmp(env, val);	
Method:	function	
Argument:	jmp_buf env;	Pointer to the area where environment is restored
	int val;	Value returned as a result of setjmp
ReturnValue:	No value is returned.	

Description:

- Restores the environment from the area indicated in "env".
- Program control is passed to the statement following that from which setjmp was called.
- The value specified in "val" is returned as the result of setjmp. However, if "val" is "0", it is converted to "1".

M

malloc

Memory Management Functions

Function: Allocates a memory area.

Format: #include <stdlib.h>

void _far * malloc(nbytes);

Method: function

Argument: size_t nbytes; Size of memory area (in bytes) to be allocated ....

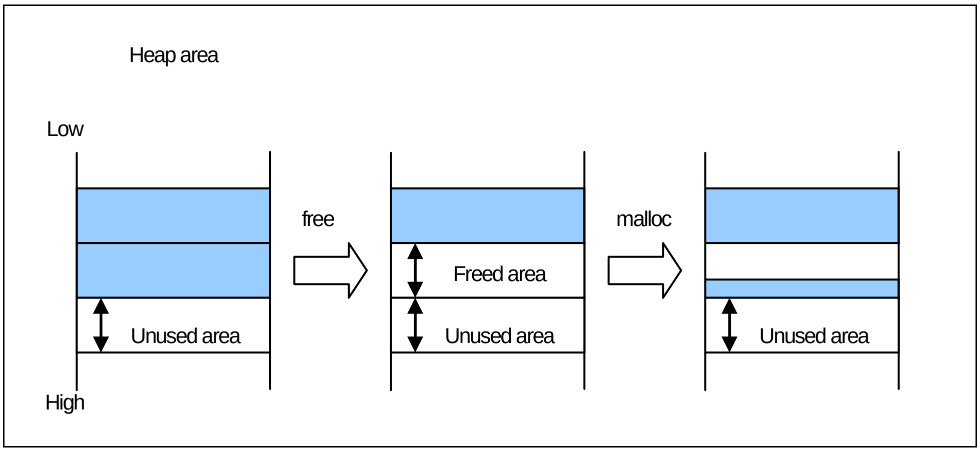
ReturnValue: Returns NULL if a memory area of the specified size could not be allocated.

Description: Dynamically allocates memory areas

Rule: malloc performs the following two checks to secure memory in the appropriate location.

(1) If memory areas have been freed with free

- If the amount of memory to be secured is smaller than that freed, the area is secured from the high address of the contiguously empty area created by free toward the low address.

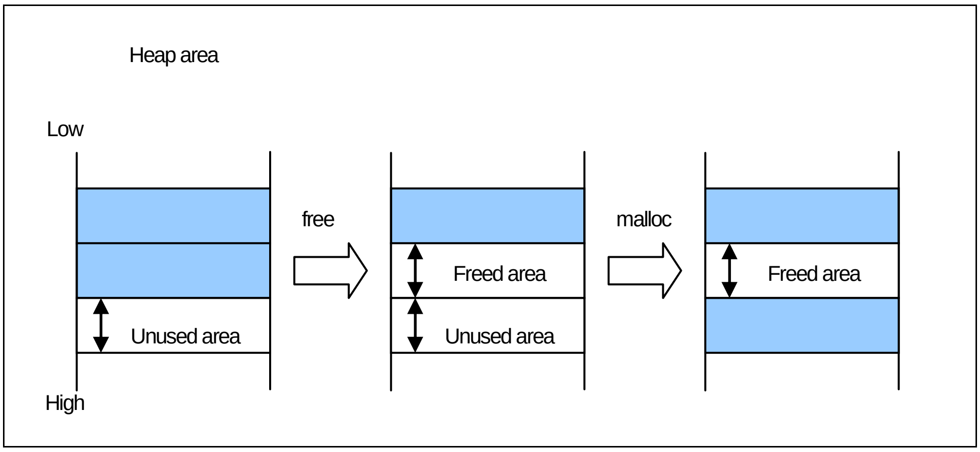


malloc

Memory Management Functions

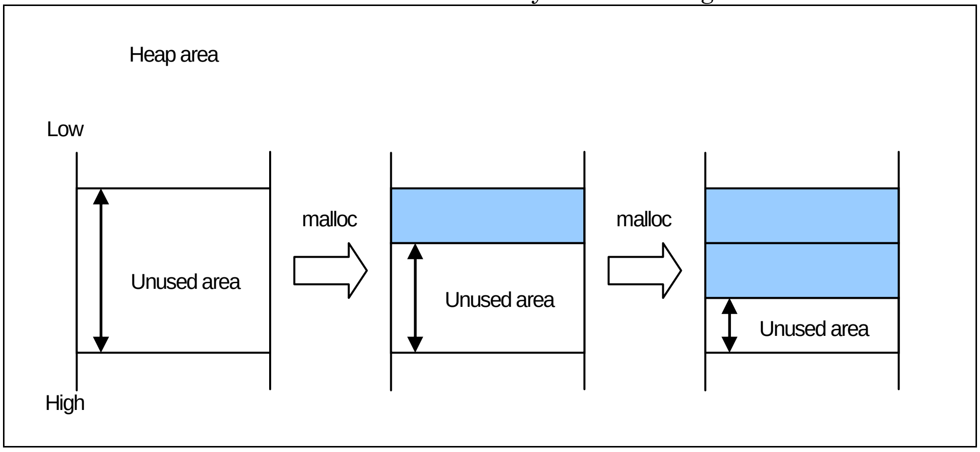
Rule:

- If the amount of memory to be secured is larger than that freed, the area is secured from the lowest address of the unused memory toward the high address.



(2) If no memory area has been freed with free

- If there is any unused area that can be secured, the area is secured from the lowest address of the unused memory toward the high address.



- If there is no unused area that can be secured, malloc returns NULL without any memory being secured.

Note:

No garbage collection is performed. Therefore, even if there are lots of small unused portions of memory, no memory is secured and malloc returns NULL unless there is an unused portion of memory that is larger than the specified size.

mblen

Multi-byte Character Multi-byte Character String Manipulate Functions

Function: Calculates the length of a multibyte character string.

Format: #include <stdlib.h>

int mblen (s,n);

Method: function

Argument:

const char _far *s;	Pointer to a multibyte character string
size_t n;	Number of searched byte

ReturnValue:
- Returns the number of bytes in the character string if 's' configures a correct multibyte character string.
- Returns -1 if 's' does not configure a correct multibyte character string.

Description: Returns 0 if 's' indicates a NULL character.

mbstowcs

Multi-byte Character Multi-byte Character String Manipulate Functions

Function: Converts a multibyte character string into a wide character string.

Format: #include <stdlib.h>

size_t mbstowcs(wcs,s,n);

Method: function

Argument:

wchar_t _far *wcs;	Pointer to an area for storing conversion wide character string
const char _far *s;	Pointer to a multibyte character string
size_t n;	Number of wide characters stored

ReturnValue:
- Returns the number of characters in the converted multibyte character string.
- Returns -1 if 's' does not configure a correct multibyte character string.

mbtowc

Multi-byte Character Multi-byte Character String Manipulate Functions

Function: Converts a multibyte character into a wide character.

Format: #include <stdlib.h>

int mbtowc(wcs,s,n);

Method: function

Argument:

wchar_t _far *wcs;	Pointer to an area for storing conversion wide character string
const char _far *s;	Pointer to a multibyte character string
size_t n;	Number of wide characters stored

ReturnValue:

- Returns the number of wide characters converted if 's' configure a correct multibyte character string.
- Returns -1 if 's' does not configure a correct multibyte character string.
- Returns 0 if 's' indicates a NULL character.

memchr

Memory Handling Functions

Function: Searches a character from a memory area.

Format: #include <string.h>

void _far * memchr(s, c, n);

Method: function

Argument:

const void _far *s;	Pointer to the memory area to be searched from
int c;	Character to be searched
size_t n;	Size of the memory area to be searched

ReturnValue:

- Returns the position (pointer) of the specified character "c" where it is found.
- Returns NULL if the character "c" could not be found in the memory area.

Description:

- Searches for the characters shown in "c" in the amount of memory specified in "n" starting at the address specified in "s".
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

memcmp

Memory Handling Functions

Function: Compares memory areas ('n' bytes).

Format: #include <string.h>

int memcmp(s1, s2, n);

Method: function

Argument:

const void _far *s1;	Pointer to the first memory area to be compared
const void _far *s2;	Pointer to the second memory area to be compared
size_t n;	Number of bytes to be compared

ReturnValue:

- Return Value= =0 — The two memory areas are equal.
- Return Value>0 — The first memory area (s1) is greater than the other.
- Return Value<0 — The second memory area (s2) is greater than the other.

Description:

- Compares each of n bytes of two memory areas
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

memcpy

Memory Handling Functions

Function: Copies n bytes of memory

Format: #include <string.h>

void _far * memcpy(s1, s2, n);

Method: function

Argument:

void _far *s1;	Pointer to the memory area to be copied to
const void _far *s2;	Pointer to the memory area to be copied from
size_t n;	Number of bytes to be copied

ReturnValue: Returns the pointer to the memory area to which the characters have been copied.

Description:

- Copies "n" bytes from memory "S2" to memory "S1".
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

memicmp

Memory Handling Functions

Function:	Compares memory areas (with alphabets handled as upper-case letters).	
Format:	#include <string.h> int memicmp(s1, s2, n);	
Method:	function	
Argument:	char _far *s1; char _far *s2; size_t n;	Pointer to the first memory area to be compared Pointer to the second memory area to be compared Number of bytes to be compared

ReturnValue:
- Return Value= =0 — The two memory areas are equal.
- Return Value>0 — The first memory area (s1) is greater than the other.
- Return Value<0 — The second memory area (s2) is greater than the other.

Description:
- Compares memory areas (with alphabets handled as upper-case letters).
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

memmove

Memory Handling Functions

Function:	Moves the area of a character string.	
Format:	#include <string.h> void _far * memmove(s1, s2, n);	
Method:	function	
Argument:	void _far *s1; const void _far *s2; size_t n;	Pointer to be moved to Pointer to be moved from Number of bytes to be moved
ReturnValue:	Returns a pointer to the destination of movement.	
Description:	If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.	

memset

Memory Handling Functions

Function: Set a memory area.

Format: #include <string.h>

void _far * memset(s, c, n);

Method: function

Argument:

void _far *s;	Pointer to the memory area to be set at
int c;	Data to be set
size_t n;	Number of bytes to be set

ReturnValue: Returns the pointer to the memory area which has been set.

Description:

- Sets "n" bytes of data "c" in memory "s".
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

modf

Mathematical Functions

Function: Calculates the division of a real number into the mantissa and exponent parts.

Format: #include <math.h>

double modf (val,pd);

Method: function

Argument:

double val;	arbitrary real number
double *pd;	Pointer to an area for storing an integer

ReturnValue: Returns the decimal part of a real number.

P

perror

Input/Output Functions

Function:	Outputs an error message to stderr.	
Format:	#include <stdio.h> void perror(s);	
Method:	function	
Argument:	const char _far *s;	Pointer to a character string attached before a message.
ReturnValue:	No value is returned.	

pow

Mathematical Functions

Function:	Calculates the power of a number.	
Format:	#include <math.h> double pow(x,y);	
Method:	function	
Argument:	double x; double y;	multiplicand power of a numbe
ReturnValue:	Returns the multiplicand x raised to the power of y.	

printf

Input/Output Functions

Function: Outputs characters with format to stdout.

Format: #include <stdio.h>

int printf(format, argument...);

Method: function

Argument: const char _far *format; Pointer of the format specifying character string

The part after the percent (%) sign in the character string given in format has the following meaning. The part between [and] is optional. Details of the format are shown below.

```
Format: %[flag][minimum field width][precision][modifier] conversion
        specification character

Example format: %-05.8ld
```

ReturnValue:

- Returns the number of characters output.
- Returns EOF if a hardware error occurs.

Description:

- Converts argument to a character string as specified in format and outputs the character string to stdout.
- When giving a pointer to argument, it is necessary to be a far type pointer.

(1) Conversion specification symbol

- d, i
  Converts the integer in the parameter to a signed decimal.
- u
  Converts the integer in the parameter to an unsigned decimal.
- o
  Converts the integer in the parameter to an unsigned octal.
- x
  Converts the integer in the parameter to an unsigned hexadecimal. Lowercase "abcdef" are equivalent to 0AH to 0FH.
- X
  Converts the integer in the parameter to an unsigned hexadecimal. Uppercase "ABCDEF" are equivalent to 0AH to 0FH.
- c
  Outputs the parameter as an ASCII character.
- s
  Converts the parameter after the string far pointer (char *) (and up to a null character '/0' or the precision) to a character string. Note that wchar_t type character strings cannot be processed.
- p
  Outputs the parameter pointer (all types) in the format 24 bits address.
- n
  Stores the number of characters output in the integer pointer of the parameter. The parameter is not converted.
- e
  Converts a double-type parameter to the exponent format. The format is [-]d.dddddde±dd.
- E
  Same as e, except that E is used in place of e for the exponent.

printf

Input/Output Functions

Description:

- f
  Converts double parameters to [-]d.dddddd format.
- g
  Converts double parameters to the format specified in e or f. Normally, f conversion, but conversion to e type when the exponent is -4 or less or the precision is less than the value of the exponent.
- G
  Same as g except that E is used in place of e for the exponent.
- −
  Left-aligns the result of conversion in the minimum field width. The default is right alignment.
- +
  Adds + or − to the result of signed conversion. By default, only the - is added to negative numbers.
- Blank' '
  By default, a blank is added before the value if the result of signed conversion has no sign.
- #
  Adds 0 to the beginning of o conversion.
  Adds 0x or 0X to the beginning when other than 0 in x or X conversion.
  Always adds the decimal point in e, E, and f conversion.
  Always adds the decimal point in g and G conversion and also outputs any 0s in the decimal place.

(2) Minimum field width

- Specifies the minimum field width of positive decimal integers.
- When the result of conversion has fewer characters than the specified field width, the left of the field is padded.
- The default padding character is the blank. However, '0' is the padding character if you specified the field with using an integer preceded by '0'.
- If you specified the − flag, the result of conversion is left aligned and padding characters (always blanks) inserted to the right.
- If you specified the asterisk (*) for the minimum field width, the integer in the parameter specifies the field width. If the value of the parameter is negative, the value after the −flag is the positive field width.

(3) Precision

Specify a positive integer after '.'. If you specify only '.' with no value, it is interpreted as zero. The function and default value differs according to the conversion type.
Floating point type data is output with a precision of 6 by default. However, no decimal places are output if you specify a precision of 0.

- d, i, o, u, x, and X conversion
  (1) If the number of columns in the result of conversion is less than the specified number, the beginning is padded with zeros.
  (2) If the specified number of columns exceeds the minimum field width, the specified number of columns takes precedence.
  (3) If the number of columns in the specified precision is less than the minimum field width the field width is processed after the minimum number of columns have bee processed.
  (4) The default is 1
  (5) Nothing is output if zero with converted by zero minimum columns.

printf

Input/Output Functions

Description:

- s conversion
  - (1) Represents the maximum number of characters.
  - (2) If the result of conversion exceeds the specified number of characters, the remainder is discarded.
  - (3) There is no limit to the number of characters in the default.
  - (4) If an asterisk (*) is used to specify precision → the integer in a parameter specifies precision.
  - (5) If the value of a parameter is negative → specification of precision has no effect.
- e, E, and f conversion

  n (where n is the precision) numerals are output after the decimal point.
- g and G conversion

  Valid characters in excess of n (where n is the precision) are not output.

(4) Qualifier

- If l, conversion of d, i, o, u, x, X, or n is performed on long int or unsigned long int parameter. If qualifier l is specified for other than conversion of d, i, o, u, x, X, or n, specification is ignored.
- If ll, conversion of d, i, o, u, x, X, or n is performed on long long or unsigned long long parameter. If qualifier ll is specified for other than conversion of d, i, o, u, x, X, or n, specification is ignored.
- If h, conversion of d, i, o, u, x, X, or n is performed on short int or unsigned short int parameter. If qualifier h is specified for other than conversion of d, i, o, u, x, X, or n, specification is ignored.
- If L, conversion of e, E, f, g, or G is performed on double parameter.

Notes:

If a new project is created for the R8C (ROM, less than 64KB) in the integrated development environment (High-performance Embedded Workshop), floating-point conversions (%e, %E, %f, %g, %G) are disabled for use, which is so designed in order to reduce the ROM size.
Also, the same applies to the r8clib.lib and r8cs16.lib libraries that come standard with the compiler package.
If necessary, create a library by the library generator without specifying -nofloat and use the generated library.

putc

Input/Output Functions

Function: Outputs one character to the stream.

Format: #include <stdio.h>

int putc(c, stream);

Method: macro

Argument:

int c;	Character to be output
FILE _far *stream;	Pointer of the stream

ReturnValue:
- Returns the output character if output normally.
- Returns EOF if an error occurs.

Description: Outputs one character to the stream.

putchar

Input/Output Functions

Function: Outputs one character to stdout.

Format: #include <stdio.h>

int putchar(c);

Method: macro

Argument:

int c;	Character to be output

ReturnValue:
- Returns the output character if output normally.
- Returns EOF if an error occurs.

Description: Outputs one character to stdout.

puts

Input/Output Functions

Function: Outputs one line to stdout.

Format: #include <stdio.h>

int puts(str);

Method: function

Argument: char _far *str; Pointer of the character string to be output

ReturnValue:
- Returns 0 if output normally.
- Returns -1 (EOF) if an error occurs.

Description:
- Outputs one line to stdout.
- The null character ('\0') at the end of the character string is replaced with the new line character('/n').

Q

qsort

Integer Arithmetic Functions

Function: Sorts elements in an array.

Format: #include <stdlib.h>

void qsort(base,nelen,size,cmp(e1,e2));

Method: function

Argument:

void _far *base;	Start address of array
size_t nelen;	Element number
size_t size;	Element size
int cmp( );	Compare function

ReturnValue: No value is returned.

Description: Sorts elements in an array.

R

rand

Integer Arithmetic Functions

Function: Generates a pseudo-random number.

Format: #include <stdlib.h>

int rand(void);

Method: function

Argument: No argument used.

ReturnValue:
- Returns the seed random number series specified in srand.
- The generated random number is a value between 0 and RAND_MAX.

realloc

Memory Management Functions

Function: Changes the size of an allocated memory area.

Format: #include <stdlib.h>

void _far * realloc(cp, nbytes);

Method: function

Argument:

void _far *cp;	Pointer to the memory area before change
size_t nbytes;	Size of memory area (in bytes) to be changed

ReturnValue:
- Returns the pointer of the memory area which has had its size changed.
- Returns NULL if a memory area of the specified size could not be secured.

Description:
- Changes the size of an area already secured using malloc or calloc.
- Specify a previously secured pointer in parameter "cp" and specify the number of bytes to change in "nbytes".

S

scanf

Input/Output Functions

Function: Reads characters with format from stdin.

Format: #include <stdio.h>
#include <ctype.h>

int scanf(format, argument...);

Method: function

Argument: const char _far *format; Pointer of format specifying character string

The part after the percent (%) sign in the character string given in format has the following meaning. The part between [and] is optional. Details of the format are shown below.

Format: %[*] [maximum field width] [qualifier] conversion specifying symbol Example format: %*5ld

ReturnValue:
- Returns the number of data entries stored in each argument.
- Returns EOF if EOF is input from stdin as data.

Description:
- Converts the characters read from stdin as specified in format and stores them in the variables shown in the arguments.
- Argument must be a far pointer to the respective variable.
- The first space character is ignored except in c and [] conversion.
- Interprets code 0x1A as the end code and ignores any subsequent data.

(1) Conversion specification symbol

- d
  Converts a signed decimal. The target parameter must be a pointer to an integer.
- i
  Converts signed decimal, octal, and hexadecimal input. Octals start with 0. Hexadecimals start with 0x or 0X. The target parameter must be a pointer to an integer.
- u
  Converts an unsigned decimal. The target parameter must be a pointer to an unsigned integer.
- o
  Converts a signed octal. The target parameter must be a pointer to an integer.
- x,X
  Converts a signed hexadecimal. Uppercase or lowercase can be used for 0AH to 0FH. The leading 0x is not included. The target parameter must be a pointer to an integer.
- s
  Stores character strings ending with the null character '\0'. The target parameter must be a pointer to a character array of sufficient size to store the character string including the null character '\0'.
  If input stops when the maximum field width is reached, the character string stored consists of the characters to that point plus the ending null character.

scanf

Input/Output Functions

Description:

- c
  Stores a character. Space characters are not skipped. If you specify 2 or more for the maximum field width, multiple characters are stored. However, the null character '\0' is not included. The target parameter must be a pointer to a character array of sufficient size to store the character string.
- p
  Converts input in the format data bank register plus offset (Example: 00:1205). The target parameter is a pointer to all types.
- [ ]
  Stores the input characters while the one or more characters between [and] are input. Storing stops when a character other than those between [and] is input. If you specify the circumflex (^) after [, only character other than those between the circumflex and] are legal input characters. Storing stops when one of the specified characters is input.
  The target parameter must be a pointer to a character array of sufficient size to store the character string including the null character '\0', which is automatically added.
- n
  Stores the number of characters already read in format conversion. The target parameter must be a pointer to an integer.
- e,E,f,g,G
  Convert to floating point format. If you specify modifier I, the target parameter must be a pointer to a double type. The default is a pointer to a float type.

(2) *(prevents data storage)

- Specifying the asterisk (*) prevents the storage of converted data in the parameter.

(3) Maximum field width

- Specify the maximum number of input characters as a positive decimal integer. In any one format conversion, the number of characters read will not exceed this number.
- If, before the specified number of characters has been read, a space character (a character that is true in function isspace()) or a character other than in the specified format is input, reading stops at that character.

(4) Qualifier

- If l, the result of conversion of d, i, o, u, x, X, or n is stored as long int or unsigned long int. Also, the result of conversion of e, E, f, g, or G is stored as double. If qualifier l is specified for other than conversion of d, i, o, u, x, X, n, e, E, f, g, or G, specification is ignored.
- If ll, the result of conversion of d, i, o, u, x, X, or n is stored as long long or unsigned long long. If qualifier l is specified for other than conversion of d, i, o, u, x, X, or n, specification is ignored.
- If h, the result of conversion of d, i, o, u, x, or X is stored as short in or unsigned short int. If qualifier h is specified for other than conversion of d, i, o, u, x, or X, specification is ignored.
- If L, the result of conversion of e, E, f, g, or G is stored as float.

setjmp

Execution Control Functions

Function:	Saves the environment before a function call
Format:	#include <setjmp.h> int setjmp(env);
Method:	function
Argument:	jmp_buf _far env; Pointer to the area where environment is saved
ReturnValue:	Returns the numeric value given by the argument of longjmp.
Description:	Saves the environment to the area specified in "env".

setlocale

Localization Functions

Function:	Sets and searches the locale information of a program.
Format:	#include <locale.h> char _far *setlocale(category,locale);
Method:	function
Argument:	int category; Locale information, search section information const char _far *locale; Pointer to a locale information character string
ReturnValue:	● Returns a pointer to a locale information character string. ● Returns NULL if information cannot be set or searched.

sin

Mathematical Functions

Function:	Calculates sine.	
Format:	#include <math.h>	
	double sin(x);	
Method:	function	
Argument:	double x;	arbitrary real number
ReturnValue:	Returns the sine of given real number x handled in units of radian.	

sinh

Mathematical Functions

Function:	Calculates hyperbolic sine.	
Format:	#include <math.h>	
	double sinh(x);	
Method:	function	
Argument:	double x;	arbitrary real number
ReturnValue:	Returns the hyperbolic sine of given real number x.	

sprintf

Input/Output Functions

Function:	Writes text with format to a character string.	
Format:	#include <stdio.h>	
	int sprintf(pointer, format, argument...);	
Method:	function	
Argument:	char _far *pointer;	Pointer of the location to be stored
	const char _far *format;	Pointer of the format specifying character string
ReturnValue:	Returns the number of characters output.	

Description:

- Converts argument to a character string as specified in format and stores them from the pointer.
- Format is specified in the same way as in printf.

sqrt

Mathematical Functions

Function: Calculates the square root of a numeric value.

Format: #include <math.h>

double sqrt(x);

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the square root of given real number x.

srand

Integer Arithmetic Functions

Function: Imparts seed to a pseudo-random number generating routine.

Format: #include <stdlib.h>

void srand(seed);

Method: function

Argument: unsigned int seed; Series value of random number

ReturnValue: No value is returned.

Description: Initializes (seeds) the pseudo random number series produced by rand using seed.

sscanf

Input/Output Functions

Function: Reads data with format from a character string.

Format: #include <stdio.h>

int sscanf(string, format, argument...);

Method: function

Argument:
const char _far *string; Pointer of the input character string
const char _far *format; Pointer of the format specifying character string

ReturnValue:
- Returns the number of data entries stored in each argument.
- Returns EOF if null character ('/0') is input as data.

Description:
- Converts the characters input as specified in format and stores them in the variables shown in the arguments.
- Argument must be a far pointer to the respective variable.
- Format is specified in the same way as in scanf.

strcat

String Handling Functions

Function: Concatenates character strings.

Format: #include <string.h>

char _far * strcat(s1, s2);

Method: function

Argument:

char _far *s1;	Pointer to the character string to be concatenated to
const char _far *s2;	Pointer to the character string to be concatenated from

ReturnValue: Returns a pointer to the concatenated character string area(s1).

Description:

- Concatenates character strings "s1" and "s2" in the sequence s1+s2[1]
- The concatenated string ends with NULL.
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strchr

String Handling Functions

Function: Searches the specified character beginning with the top of the character string.

Format: #include <string.h>

char _far * strchr(s, c);

Method: function

Argument:

const char _far *s;	Pointer to the character string to be searched in
int c;	Character to be searched for

ReturnValue:

- Returns the position of character "c" that is first encountered in character string "s."
- Returns NULL when character string "s" does not contain character "c".

Description:

- Searches for character "c" starting from the beginning of area "s".
- You can also search for '\0'.
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

[1] There must be adequate space to accommodate s1 plus s2.

strcmp

String Handling Functions

Function: Compares character strings.

Format: #include <string.h>

int strcmp(s1, s2);

Method: macro,function

Argument:

	const char _far *s1;	Pointer to the first character string to be compared
	const char _far *s2;	Pointer to the second character string to be compared

ReturnValue:

- ReturnValue= =0 The two character strings are equal.
- ReturnValue>0 The first character string (s1) is greater than the other.
- ReturnValue<0 The second character string (s2) is greater than the other.

Description:

- Usually, the program code described by macro is used for this function. In using the function in a library, please describe it as #undef strcmp after description of #include <string.h>.
- Compares each byte of two character strings ending with NULL
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strcoll

String Handling Functions

Function: Compares character strings (using locale information).

Format: #include <string.h>

int strcoll(s1, s2);

Method: function

Argument:

	const char _far *s1;	Pointer to the first character string to be compared
	const char _far *s2;	Pointer to the second character string to be compared

ReturnValue:

- ReturnValue= =0 The two character strings are equal
- ReturnValue>0 The first character string (s1) is greater than the other
- ReturnValue<0 The second character string (s2) is greater than the other

Description: If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strcpy

String Handling Functions

Function: Copies a character string.

Format: #include <string.h>

char _far * strcpy(s1, s2);

Method: function

Argument:

char _far *s1;	Pointer to the character string to be copied to
const char _far *s2;	Pointer to the character string to be copied from

ReturnValue: Returns a pointer to the character string at the destination of copy.

Description:

- Copies character string "s2" (ending with NULL) to area "s1"
- After copying, the character string ends with NULL.
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, functions may be expanded in-line by optimization.

strcspn

String Handling Functions

Function: Calculates the length (number) of unspecified characters that are not found in the other character string

Format: #include <string.h>

size_t strcspn(s1, s2);

Method: function

Argument:

const char _far *s1;	Pointer to the character string to be searched in
const char _far *s2;	Pointer to the character string to be searched for

ReturnValue: Returns the length (number) of unspecified characters.

Description:

- Calculates the size of the first character string consisting of characters other than those in 's2' from area 's1', and searches the characters from the beginning of 's1'.
- You cannot search for '\0'.

stricmp

String Handling Functions

Function: Compares character strings. (All alphabets are handled as upper-case letters.)

Format: #include <string.h>

int stricmp(s1, s2);

Method: function

Argument:

char _far *s1;	Pointer to the first character string to be compared
char _far *s2;	Pointer to the second character string to be compared

ReturnValue:

- ReturnValue= =0 The two character strings are equal.
- ReturnValue>0 The first character string (s1) is greater than the other.
- ReturnValue<0 The second character string (s2) is greater than the other.

Description: Compares each byte of two character strings ending with NULL. However, all letters are treated as uppercase letters.

strerror

String Handling Functions

Function: Converts an error number into a character string.

Format: #include <string.h>

char _far * strerror(errcode);

Method: function

Argument: int errcode; error code

ReturnValue: Returns a pointer to a message character string for the error code.

Description: stderr returns the pointer for a static array.

strlen

String Handling Functions

Function:	Calculates the number of characters in a character string.	
Format:	#include <string.h>	
	size_t strlen(s);	
Method:	function	
Argument:	const char _far *s;	Pointer to the character string to be operated on to calculate length
ReturnValue:	Returns the length of the character string.	
Description:	Determines the length of character string "s" (to NULL).	

strncat

String Handling Functions

Function:	Concatenates character strings ('n' characters).	
Format:	#include <string.h>	
	char _far * strncat(s1, s2, n);	
Method:	function	
Argument:	char _far *s1;	Pointer to the character string to be concatenated to
	const char _far *s2;	Pointer to the character string to be concatenated from
	size_t n;	Number of characters to be concatenated
ReturnValue:	Returns a pointer to the concatenated character string area.	

Description:

- Concatenates character strings "s1" and "n" characters from character string "s2".
- The concatenated string ends with NULL.
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strncmp

String Handling Function

Function: Compares character strings ('n' characters).

Format: #include <string.h>

```
int strncmp( s1, s2, n );
```

Method: function

Argument:

const char _far *s1;	Pointer to the first character string to be compared
const char _far *s2;	Pointer to the second character string to be compared
size_t n;	Number of characters to be compared

ReturnValue:

- ReturnValue= =0 The two character strings are equal.
- ReturnValue>0 The first character string (s1) is greater than the other.
- ReturnValue<0 The second character string (s2) is greater than the other.

Description:

- Compares each byte of n characters of two character strings ending with NULL.
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strncpy

String Handling Function

Function: Copies a character string ('n' characters).

Format: #include <string.h>

```
char _far * strncpy( s1, s2, n );
```

Method: function

Argument:

char _far *s1;	Pointer to the character string to be copied to
const char _far *s2;	Pointer to the character string to be copied from
size_t n;	Number of characters to be copied

ReturnValue: Returns a pointer to the character string at the destination of copy.

Description:

- Copies "n" characters from character string "s2" to area "s1". If character string "s2" contains more characters than specified in "n", they are not copied and '\0' is not appended. Conversely, if "s2" contains fewer characters than specified in "n", '\0's are appended to the end of the copied character string to make up the number specified in "n".
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strnicmp

String Handling Functions

Function: Compares character strings ('n' characters). (All alphabets are handled as uppercase letters.)

Format: #include <string.h>

int strnicmp(s1, s2, n);

Method: function

Argument:

char _far *s1;	Pointer to the first character string to be compared
char _far *s2;	Pointer to the second character string to be compared
size_t n;	Number of characters to be compared

ReturnValue:

- ReturnValue= =0 The two character strings are equal.
- ReturnValue>0 The first character string (s1) is greater than the other.
- ReturnValue<0 The second character string (s2) is greater than the other.

Description:

- Compares each byte of n characters of two character strings ending with NULL.However, all letters are treated as uppercase letters.
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strpbrk

String Handling Functions

Function: Searches the specified character in a character string from the other character string.

Format: #include <string.h>

char _far * strpbrk(s1, s2);

Method: function

Argument:

const char _far *s1;	Pointer to the character string to be searched in
const char _far *s2;	Pointer to the character string of the character to be searched for

ReturnValue:

- Returns the position (pointer) where the specified character is found first.
- Returns NULL if the specified character cannot be found.

Description:

- Searches the specified character "s2" from the other character string in "s1" area.
- You cannot search for '\0'.
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strrchr

String Handling Functions

Function: Searches the specified character from the end of a character string.

Format: #include <string.h>

char _far * strrchr(s, c);

Method: function

Argument:

const char _far *s;	Pointer to the character string to be searched in
int c;	Character to be searched for

ReturnValue:

- Returns the position of character "c" that is last encountered in character string "s."
- Returns NULL when character string "s" does not contain character "c".

Description:

- Searches for the character specified in "c" from the end of area "s".
- You can search for '\0'.
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strspn

String Handling Functions

Function: Calculates the length (number) of specified characters that are found in the character string.

Format: #include <string.h>

size_t strspn(s1, s2);

Method: function

Argument:

const char _far *s1;	Pointer to the character string to be searched in
const char _far *s2;	Pointer to the character string of the character to be searched for

ReturnValue: Returns the length (number) of specified characters.

Description:

- Calculates the size of the first character string consisting of characters in 's2' from area 's1', and searches the characters from the beginning of 's1'.
- You cannot search for '\0'.
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strstr

String Handling Functions

Function: Searches the specified character from a character string.

Format: #include <string.h>

char _far * strstr(s1, s2);

Method: function

Argument:

const char _far *s1;	Pointer to the character string to be searched in
const char _far *s2;	Pointer to the character string of the character to be searched for

ReturnValue:
- Returns the position (pointer) where the specified character is found.
- Returns NULL when the specified character cannot be found.

Description:
- Returns the location (pointer) of the first character string "s2" from the beginning of area "s1".
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strtod

Character String Value Convert Functions

Function: Converts a character string into a double-type integer.

Format: #include <stdlib.h>

double strtod(s,endptr);

Method: function

Argument:

const char _far *s;	Pointer to the converted character string
char _far * _far *endptr;	Pointer to the remaining character strings that have not been converted

ReturnValue:

ReturnValue = = 0L	Does not constitute a number.
ReturnValue != 0L	Returns the configured number in double type.

Description: If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strtok

String Handling Functions

Function: Divides some character string from a character string into tokens.

Format: #include <string.h>

```
char _far * strtok( s1, s2 );
```

Method: function

Argument:

char _far *s1;	Pointer to the character string to be divided up
const char _far *s2;	Pointer to the punctuation character to be divided with

ReturnValue:

- Returns the pointer to the divided token when character is found.
- Returns NULL when character cannot be found.

Description:

- In the first call, returns a pointer to the first character of the first token. A NULL character is written after the returned character. In subsequent calls (when "s1" is NULL), this instruction returns each token as it is encountered. NULL is returned when there are no more tokens in "s1".
- If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strtol

Character String Value Convert Function

Function: Converts a character string into a long-type integer.

Format: #include <stdlib.h>

```
long strtol( s,endptr,base );
```

Method: function

Argument:

const char _far *s;	Pointer to the converted character string
char _far * _far *endptr;	Pointer to the remaining character strings that have not been converted.
int base;	Base of values to be read in (0 to 36) Reads the format of integral constant if the base of value is zero

ReturnValue:

- ReturnValue = = 0L Does not constitute a number.
- ReturnValue != 0L Returns the configured number in long type.

Description: If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strtoul

Character String Value Convert Function

Function: Converts a character string into an unsigned long-type integer.

Format: #include <stdlib.h>

unsigned long strtoul(s,endptr,base);

Method: function

Argument:

const char _far *s;	Pointer to the converted character string
char _far * _far *endptr;	Pointer to the remaining character strings that have not been converted.
int base;	Base of values to be read in (0 to 36) Reads the format of integral constant if the base of value is zero

ReturnValue:

- ReturnValue = = 0L Does not constitute a number.
- ReturnValue != 0L Returns the configured number in long type.

Description: If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

strxfrm

Character String Value Convert Functions

Function: Converts a character string (using locale information).

Format: #include <string.h>

size_t strxfrm(s1,s2,n);

Method: function

Argument:

char _far *s1;	Pointer to an area for storing a conversion result character string.
const char _far *s2;	Pointer to the character string to be converted.
size_t n;	Number of bytes converted

ReturnValue: Returns the number of characters converted.

Description: If the option -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, another function, etc. that has better code efficiency may be selected by optimization.

T

tan

Mathematical Functions

Function: Calculates tangent.

Format: #include <math.h>

double tan(x);

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the tangent of given real number x handled in units of radian.

tanh

Mathematical Functions

Function: Calculates hyperbolic tangent.

Format: #include <math.h>

double tanh(x);

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the hyperbolic tangent of given real number x.

tolower

Character Handling Functions

Function: Converts the character from an upper-case to a lower-case.

Format: #include <ctype.h>

int tolower(c);

Method: macro

Argument: int c; Character to be converted

ReturnValue:
- Returns the lower-case letter if the argument is an upper-case letter.
- Otherwise, returns the passed argument as is.

Description: Converts the character from an upper-case to a lower-case.

toupper

Character Handling Functions

Function: Converts the character from a lower-case to an upper-case.

Format: int toupper(c);

Method: macro

Argument: int c; Character to be converted

ReturnValue:
- Returns the upper-case letter if the argument is a lower-case letter.
- Otherwise, returns the passed argument as is.

Description: Converts the character from a lower-case to an upper-case.

U

ungetc

Input/Output Functions

Function: Returns one character to the stream

Format: #include <stdio.h>

int ungetc(c, stream);

Method: macro

Argument:

int c;	Character to be returned
FILE _far *stream;	Pointer of stream

ReturnValue:

- Returns the returned one character if done normally.
- Returns EOF if the stream is in write mode, an error or EOF is encountered, or the character to be sent back is EOF.

Description:

- Returns one character to the stream.
- Interprets code 0x1A as the end code and ignores any subsequent data.

V

vfprintf

Input/Output Functions

Function: Output to a stream with format.

Format: #include <stdarg.h>
#include <stdio.h>

int vfprintf(stream, format, ap);

Method: function

Argument:

FILE _far *stream;	Pointer of stream
const char _far *format;	Pointer of the format specifying character string
va_list ap;	Pointer of argument list

ReturnValue: Returns the number of characters output.

Description:
- Output to a stream with format.
- When writing pointers in variable-length variables, make sure they are a far-type pointer.

vprintf

Input/Output Functions

Function: Output to stdout with format.

Format: #include <stdarg.h>
#include <stdio.h>

int vprintf(format, ap);

Method: function

Argument:

const char _far *format;	Pointer of the format specifying character string
valist ap;	Pointer to the top of a parameter list

ReturnValue: Returns the number of characters output.

Description:
- Output to stdout with format.
- When writing pointers in variable-length variables, make sure they are a far-type pointer.

vsprintf

Input/Output Functions

Function: Output to a buffer with format.

Format: #include <stdarg.h>
#include <stdio.h>

int vfprintf(s, format, ap);

Method: function

Argument:

char _far *s;	Pointer of the location to be store
const char _far *format;	Pointer of the format specifying character string
va_list ap;	Pointer of argument list

ReturnValue: Returns the number of characters output.

Description: When writing pointers in variable-length variables, make sure they are a far-type pointer.

W

wcstombs

Multi-byte Character Multi-byte Character String Manipulate Functions

Function:	Converts a wide character string into a multibyte character string.	
Format:	#include <stdlib.h> size_t _wcstombs(s, wcs, n);	
Method:	function	
Argument:	char _far *s;	Pointer to an area for storing conversion multibyte character string
	const wchar_t _far *wcs;	Pointer to a wide character string
	size_t n;	Number of wide characters stored

ReturnValue:

- Returns the number of stored multibyte characters if the character string was converted correctly.
- Returns -1 if the character string was not converted correctly.

wctomb

Multi-byte Character Multi-byte Character String Manipulate Functions

Function:	Converts a wide character into a multibyte character.	
Format:	#include <stdlib.h> int wctomb(s,wchar);	
Method:	function	
Argument:	char _far *s;	Pointer to an area for storing conversion multibyte character string
	wchar_t wchar;	wide character

ReturnValue:

- Returns the number of bytes contained in the multibyte characters.
- Returns -1 if there is no corresponding multibyte character.
- Returns 0 if the wide character is 0.

E.2.4 Using the Standard Library

a. Notes on Regarding Standard Header File

When using functions in the standard library, always be sure to include the specified standard header file. If this header file is not included, the integrity of arguments and return values will be lost, making the program unable to operate normally.

b. Notes on Regarding Optimization of Standard Library

If one of the optimization options -O[3–5], -OR, -OR_MAX(-ORM), -OS, or -OS_MAX(-OSM) is specified, optimization on standard functions is performed. This optimization can be inhibited by specifying -Ono_stdlib. To use some function with the same name as one of the standard library functions as a user function, inhibit this optimization.

(1) Inline padding of functions

Regarding functions strcpy and memcpy, the system performs inline padding of functions if the conditions inTable E.13 are met.

Table E.13 Optimization Conditions for Standard Library Functions

Function Name	Optimization Condition	Description Example
strcpy	First argument:near pointer Second argument:string constant	strcpy(str, "sample");
memcpy	First argument:near pointer Second argument: far pointer Third argument:constant	memcpy(str ,"sample", 6); memcpy(str , fp, 6);

E.3 Modifying Standard Library

The NC30 package includes a sophisticated function library which includes functions such as the scanf and printf I/O functions. These functions are normally called high-level I/ O functions. These high-level I/O functions are combinations of hardware-dependent lowlevel I/O functions.
In M16C/80 series application programs, the I/O functions may need to be modified according to the target system's hardware. This is accomplished by modifying the source file for the standard library.
This chapter describes how to modify the NC30 standard library to match the target system.
The entry vedrsion does not come with source files for the standard function library. Therefore, the standard function library cannot be customized for the entry version.

E.3.1 Structure of I/O Functions

As shown in Figure E.1,the I/O functions work by calling lower-level functions (level 2 . level 3) from the level 1 function. For example, fgets calls level 2 fgetc, and fgetc calls a level 3 function.
Only the lowest level 3 functions are hardware-dependent (I/O port dependent) in the Micro Processor. If your application program uses an I/O function, you may need to modify the source files for the level 3 functions to match the system.

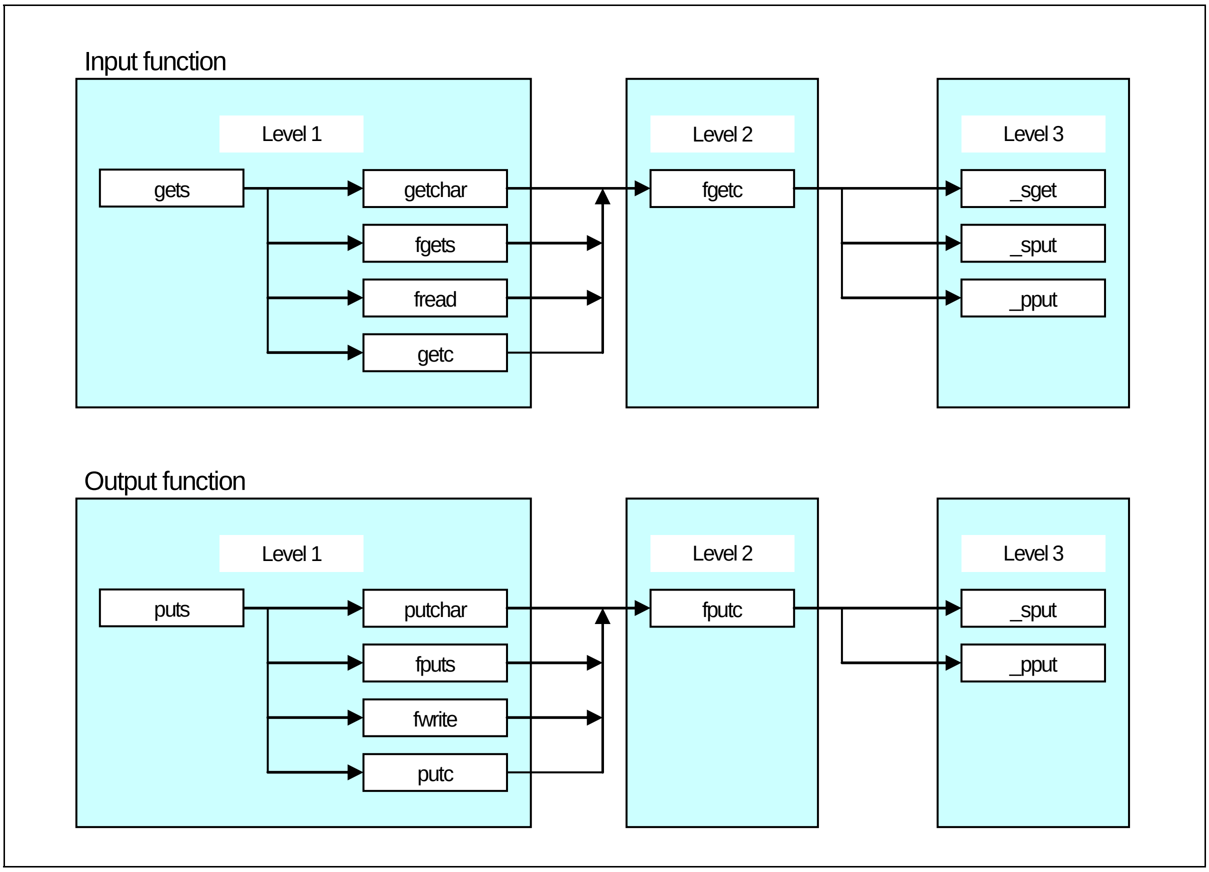


Figure E.1 Calling Relationship of I/O Functions

E.3.2 Sequence of Modifying I/O Functions

Figure E.2 outlines how to modify the I/O functions to match the target system.

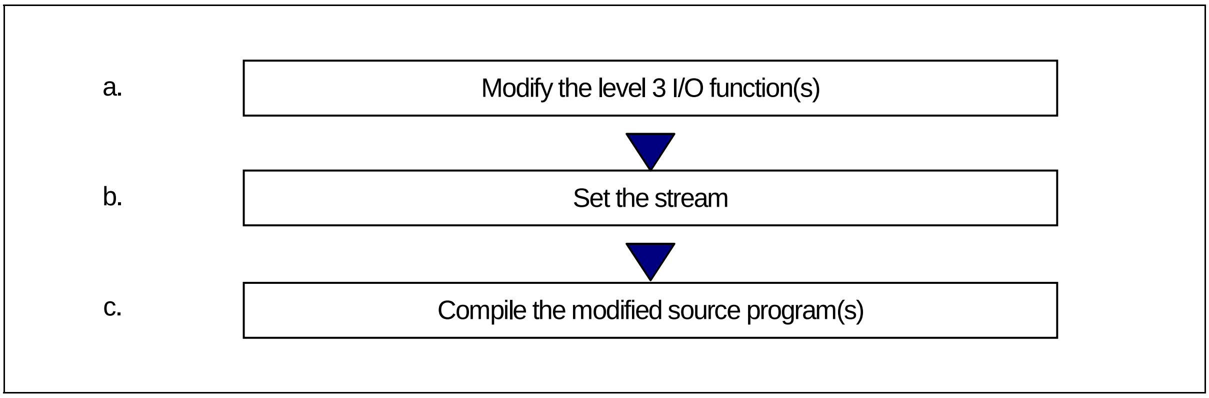


Figure E.2 Example Sequence of Modifying I/O Functions

a. Modifying Level 3 I/O Function

Level-3 input/output functions are the one that performs 1-byte input/output to and from the M16C series or R8C family input/output ports. The level-3 input/output functions include _sget and _sput that perform input/output to and from the serial communication circuit (UART) and _pput that performs input/output to and from the Centronics communication circuit.

(1) Circuit settings

- Processor mode: Microprocessor mode
- Clock frequency: 20MHz
- External bus size: 16 bits

(2) Initial serial communications settings

- Use UART1
- Baud rate: 9600bps
- Data size: 8 bits
- Parity: None
- Stop bits: 2 bits

* Initial settings for these serial communications are made in the _init function.

The level-3 input/output functions are written in the C source file "device.c." Specifications of the level-3 input/output functions are shown in Table E.14.

Table E.14 Specifications of Level 3 Functions

Input functions	Parameters	Return value (int type)
_sget	None.	If no error occurs, returns the input character Returns EOF if an error occurs

Output unctions	Parameters(int type)	Return value (int type)
_sput _pput	Character to output	If no error occurs, returns 1 Returns EOF if an error occurs

Serial communications are set in one of the two UARTs that the M16C series and R8C family have, or UART1. The device.c is written so as to allow the selection of UART0 with a conditional compile command. Here is the method.

- To use UART0, write #define __UART0__ 1 at the beginning of the device.c file, or
- To use UART0, specify -D__UART0__ at compile time.

To use both UARTs, modify the file as follows:

(1) Delete the conditional compiling commands from the beginning of the device.c file.

(2) Alter the special register name of UART0 defined with #pragma ADDRESS to a variable different than UART1 by rewriting it.

(3) Reproduce the level 3 functions _sget and _sput for UART0 and change them to different variable names such as _sget0 and _sput0.

(4) Also reproduce the speed function for UART0 and change the function name to something like speed0.

This completes modification of device.c.

Next, alter the _init function by which the input/output functions are initialized to change stream settings. How to set streams is described in the next section.

b. Stream Settings

The NC30 standard library has five items of stream data (stdin, stdout, stderr, stdaux, and stdprn) as external structures. These external structures are defined in the standard header file stdio.h and control the mode information of each stream (flag indicating whether input or output stream) and status information (flag indicating error or EOF).

Table E.15 Stream Information

Stream information	Name
stdin	Standard input
stdout	Standard output
stderr	Standard error output (error is output to stdout)
stdaux	Standard auxiliary I/O
stdprn	Standard printer output

The stream corresponding to the NC30 standard library functions shown shaded in Figure E.3 are fixed to standard input (stdin) and standard output (stdout). The stream cannot be changed for these functions. The output direction of stderr is defined as stdout in #define.

The stream can only be changed for functions that specify pointers to the stream as parameters such as fgetc and fputc.

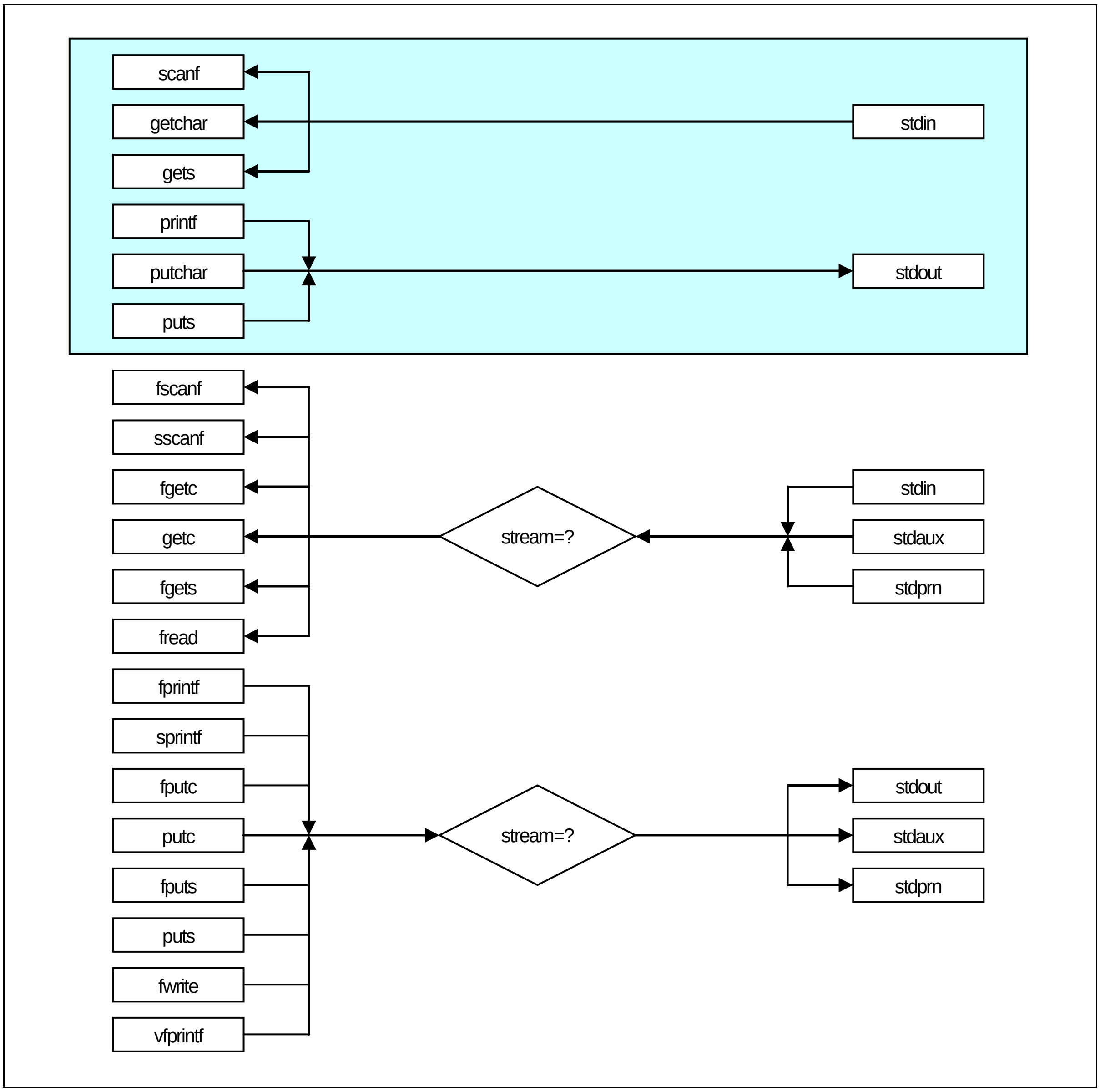


Figure E.3 Relationship of Functions and Streams

Figure E.4 shows the stream definition in stdio.h.

```
/*******************************************************************************
*
* standard I/O header file
            :
        (omitted)
            :
typedef struct _iobuf {
        char    _buff;                      /* Store buffer for ungetc */             ← [1]
        int     _cnt;                       /* Strings number in _buff(1 or 0) */     ← [2]
        int     _flag;                      /* Flag */                                ← [3]
        int     _mod;                       /* Mode */                                ← [4]
        int     (*_func_in)(void);   /* Pointer to one byte input function */         ← [5]
        int     (*_func_out)(int);   /* Pointer to one byte output function */        ← [6]
} FILE;
#define _IOBUF_DEF
            :
        (omitted)
            :
extern FILE _iob[];
#define stdin   (&_iob[0])  /* Fundamental input */
#define stdout  (&_iob[1])  /* Fundamental output */
#define stdaux  (&_iob[2])  /* Fundamental auxialiary input output */
#define stdprn  (&_iob[3])  /* Fundamental printer output */

#define stderr  stdout              /* NC no-support */

/***************************************************************
*
****************************************************************/
#define _IOREAD 1       /* Read only flag */
#define _IOWRT  2       /* Write only flag */
#define _IOEOF  4       /* End of file flag */
#define _IOERR  8       /* Error flag */
#define _IORW   16      /* Read and write flag */
#define _NFILE  4       /* Stream number */
#define _TEXT   1       /* Text mode flag */
#define _BIN    2       /* Binary mode flag */

        (remainder omitted)
            :
```

Figure E.4 Stream Definition in stdio.h

Let's look at the elements of the file structures shown in Figure E.4. Items [1] to [6] correspond to [1] to [6] in Figure E.4

(1) char _buff

Functions scanf and fscanf read one character ahead during input. If the character is no use, function ungetc is called and the character is stored in this variable.

If data exists in this variable, the input function uses this data as the input data.

(2) int _cnt

Stores the _buff data count (0 or 1)

(3) int _flag

Stores the read-only flag (_IOREAD), the write-only flag (_IOWRT), the read-write flag (_IORW), the end of file flag (_IOEOF) and the error flag (_IOERR).

- _IOREAD,_IOWRT,_IORW

  These flags specify the stream operating mode. They are set during stream initialization.

- _IOEOF,_IOERR

  These flags are set according to whether an EOF is encountered or error occurs in the I/O function.

(4) int _mod

Stores the flags indicating the text mode (_TEXT) and binary mode (_BIN).

- Text mode

  Echo-back of I/O data and conversion of characters. See the source programs (fgetc.c and fputc.c) of the fgetc and fputc functions for details of echo back and character conversion.

- Binary mode

  No conversion of I/O data. These flags are set in the initialization block of the stream.

(5) int (*_func_in)(void)

When the stream is in read-only mode (_IOREAD) or read/write mode (_IORW), stores the level 3 input function pointer. Stores a NULL pointer in other cases.

This information is used for indirect calling of level 3 input functions by level 2 input functions.

(6) int (*_func_out)(void)

When the stream is in write mode (_IOWRT), stores the level 3 output function pointer. If the stream can be input (_IOREAD or _IORW), and is in text mode, it stores the level 3 output function pointer for echo back. Stores a NULL pointer in other cases.

This information is used for indirect calling of level 3 output functions by level 2 output functions.

To initialize streams, set values for all elements but char_buff.

The function _init is used to initialize streams. The function _init is called from the startup program ncrt0.a30 or resetpreg.c.

Figure E.5 shows the source program of the _init function.

```
#include <stdio.h>

FILE _iob[4];

void _init( void );

void _init( void )
{
        stdin->_cnt = 0;
        stdout->_cnt = 0;
        stdaux->_cnt = 0;
        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 : default */
        speed(_96, _B8, _PN, _S2);
#endif
}
```

Figure E.5 Source file of init function (init.c)

For a system that uses the two UARTs of the M16C series and R8C family, alter the _init function following the procedure described below. In the preceding section, we've set the functions for UART0 temporarily as _sget0, _sput0, and speed0 in the device.c source file.

(1) Use the standard auxiliary I/O (stdaux) for the UART0 stream.
(2) Set the flag (_flag) and mode (_mod) for standard auxiliary I/O to match the system.
(3) Set the level 3 function pointer for standard auxiliary I/O.
(4) Delete the conditional compile commands for the speed function and change to function speed0 for UART0.

These settings allow both UARTs to be used. However, functions using the standard I/O stream cannot be used for standard auxiliary I/O used by UART0. Therefore, only use functions that take streams as parameters. Figure E.6 shows how to change the _init function.

```
void _init ( void )
{
            :
        (omitted)
            :
    stdaux->_flag = _IORW;              ← [2](set read/write mode)
            :
        (omitted)
            :
    stdaux->_mod = _TEXT;               ← [2](set text mode)
            :
        (omitted)
            :
    stdaux->_func_in = _sget0;          ← [3](set UART0 level 3 input function)
            :
        (omitted)
            :
    stdaux->_func_out = _sput0;         ← [3](set UART0 level 3 input function)
            :
        (omitted)
            :
    speed(_96, _B8, _PN, _S2);          ← [4](set UART0 speed function)

}
```

Figure E.6 Modifying the init Function

c. Incorporating the Modified Source Program

Specify the source file of the altered functions when linking the object files. In this case, the functions specified at link time become effective, so that the functions with the same names as in the library file are not included.An example is shown in Figure E.7

```
% nc30   -c   -g   -osample   ncrt0.a30   device.obj   init.obj   sample.c<RET>

* This example shows the command line when device.c and init.c are modified.
```

Figure E.7 Method of Directly Linking Modified Source Programs

E.4 EC++ Class Libraries

(1) Overview of Libraries

This section describes the specifications of the EC++ class libraries, which can be used as standard libraries in C++ programs. The class library types and corresponding standard include files are described. The specifications of each class library are given in accordance with the library configuration.

- Library types

  Table E.16 shows the class library types and the corresponding standard include files.

Table E.16 Class Library Types and Corresponding Standard Include Files

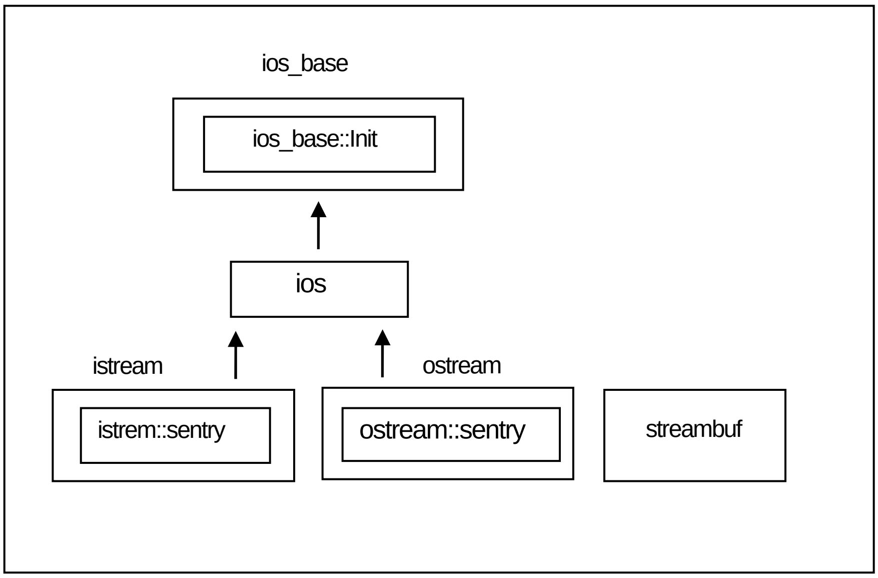
Library Type	Description	Standard Include Files
Stream input/output class library	Performs input/output processing	<ios>, <streambuf>, <istream>, <ostream>, <iostream>, <iomanip>
Memory management library	Performs memory allocation and deallocation	<new>
Complex number calculation class library	Performs calculation of complex number data	<complex>
String manipulation class library	Performs string manipulation	<string>

(2) Stream Input/Output Class Library

The header files for stream input/output class libraries are as follows:

- <ios>
  Defines data members and function members that specify input/output formats and manage the input/output states. The <ios> header file also defines the Init and ios_base classes in addition to the ios class.
- <streambuf>
  Defines functions for the stream buffer.
- <istream>
  Defines input functions from the input stream.
- <ostream>
  Defines output functions to the output stream.
- <iostream>
  Defines input/output functions.
- <iomanip>
  Defines manipulators with parameters.

The following shows the inheritance relation of the above classes. An arrow (->) indicates that a derived class references a base class. The **streambuf** class has no inheritance relation.



If stream manipulation on files is required, a class for manipulating buffers on files needs to be implemented.
To create it, mystrbuf will prove helpful.
Because the current implementation of mystrbuf is "un-buffered," the interface given below needs to be implemented as appropriate for the user system.
open(), close(), setvbuf(), seek(), ftell()

The following types are used by stream input/output class libraries.

Type	Definition Name	Description
Type	streamoff	Defined as **long** type
	streamsize	Defined as **long** type
	int_type	Defined as **long** type
	pos_type	Defined as **long** type
	off_type	Defined as **long** type

(a) ios_base::Init Class

Type	Definition Name	Description
Variable	init_cnt	This is the static data member to count the number of stream input/output objects.
Function	Init()	Constructor
	~Init()	Destructor

1. ios_base::Init::Init()
   Constructor of class **Init**.
   Increments **init_cnt**.

2. ios_base::Init::~Init()
   Destructor of class **Init**.
   Decrements **init_cnt**.

(b) ios_base Class

Type	Definition Name	Description
Type	fmtflags	Type that indicates the format control information
	iostate	Type that indicates the stream buffer input/output state
	openmode	Type that indicates the open mode of the file
	seekdir	Type that indicates the seek state of the stream buffer
Variable	fmtfl	Format flag
	wide	Field width
	prec	Precision (number of decimal point digits) at output
	fillch	Fill character
Function	void _ec2p_init_base()	Initializes the base class
	void _ec2p_copy_base(ios_base _far &ios_base_dt)	Copies **ios_base_dt**
	ios_base()	Constructor
	~ios_base()	Destructor
	fmtflags flags() const	References the format flag (**fmtfl**)
	fmtflags flags(fmtflags fmtflg)	Sets **fmtflg**&format flag (**fmtfl**) to the format flag (**fmtfl**)
	fmtflags setf(fmtflags fmtflg)	Sets **fmtflg** to format flag (**fmtfl**)
	fmtflags setf(fmtflags fmtflg, fmtflags mask)	Sets **mask**&**fmtflg** to the format flag (**fmtfl**)
	void unsetf(fmtflags mask)	Sets ~**mask**&format flag (**fmtfl**) to the format flag (**fmtfl**)
	char fill() const	References the fill character (**fillch**)
	char fill(char ch)	Sets **ch** as the fill character (**fillch**)
	int precision() const	References the precision (**prec**)
	streamsize precision(streamsize preci)	Sets **preci** as precision (**prec**)
	streamsize width() const	References the field width (**wide**)
	streamsize width(streamsize wd)	Sets **wd** as field width (**wide**)

1. ios_base::fmtflags
   Defines the format control information relating to input/output processing.
   The definition for each bit mask of **fmtflags** is as follows:

const ios_base::fmtflags ios_base::boolalpha	= 0x0000;
const ios_base::fmtflags ios_base::skipws	= 0x0001;
const ios_base::fmtflags ios_base::unitbuf	= 0x0002;
const ios_base::fmtflags ios_base::uppercase	= 0x0004;
const ios_base::fmtflags ios_base::showbase	= 0x0008;
const ios_base::fmtflags ios_base::showpoint	= 0x0010;
const ios_base::fmtflags ios_base::showpos	= 0x0020;
const ios_base::fmtflags ios_base::left	= 0x0040;
const ios_base::fmtflags ios_base::right	= 0x0080;
const ios_base::fmtflags ios_base::internal	= 0x0100;
const ios_base::fmtflags ios_base::adjustfield	= 0x01c0;
const ios_base::fmtflags ios_base::dec	= 0x0200;
const ios_base::fmtflags ios_base::oct	= 0x0400;
const ios_base::fmtflags ios_base::hex	= 0x0800;
const ios_base::fmtflags ios_base::basefield	= 0x0e00;
const ios_base::fmtflags ios_base::scientific	= 0x1000;
const ios_base::fmtflags ios_base::fixed	= 0x2000;
const ios_base::fmtflags ios_base::floatfield	= 0x3000;
const ios_base::fmtflags ios_base::_fmtmask	= 0x3fff;

2. ios_base::iostate
   Defines the input/output state of the stream buffer.
   The definition for each bit mask of **iostate** is as follows:

const ios_base::iostate ios_base::goodbit	= 0x0;
const ios_base::iostate ios_base::eofbit	= 0x1;
const ios_base::iostate ios_base::failbit	= 0x2;
const ios_base::iostate ios_base::badbit	= 0x4;
const ios_base::iostate ios_base::_statemask	= 0x7;

3. ios_base::openmode
   Defines open mode of the file.
   The definition for each bit mask of **openmode** is as follows:

const ios_base::openmode ios_base::in	= 0x01;	Opens the input file.
const ios_base::openmode ios_base::out	= 0x02;	Opens the output file.
const ios_base::openmode ios_base::ate	= 0x04;	Seeks for **eof** only once after the file has been opened.
const ios_base::openmode ios_base::app	= 0x08;	Seeks for **eof** each time the file is written to.
const ios_base::openmode ios_base::trunc	= 0x10;	Opens the file in overwrite mode.
const ios_base::openmode ios_base::binary	= 0x20;	Opens the file in binary mode.

4. ios_base::seekdir
   Defines the seek state of the stream buffer.
   Determines the position in a stream to continue the input/output of data.
   The definition for each bit mask of **seekdir** is as follows:

```
const ios_base::seekdir ios_base::beg     = 0x0;
const ios_base::seekdir ios_base::cur     = 0x1;
const ios_base::seekdir ios_base::end     = 0x2;
```

5. void ios_base::_ec2p_init_base()
   The initial settings are as follows:
   fmtfl = skipws | dec;
   wide = 0;
   prec = 6;
   fillch = ' ';

6. void ios_base::_ec2p_copy_base(ios_base _far & ios_base_dt)
   Copies **ios_base_dt**.

7. ios_base::ios_base()
   Constructor of class **ios_base**.
   Calls **Init::Init()**.

8. ios_base::~ios_base()
   Destructor of class **ios_base**.

9. ios_base::fmtflags ios_base::flags() const
   References the format flag (**fmtfl**).
   Return value:
   Format flag (**fmtfl**)

10. ios_base::fmtflags ios_base::flags(fmtflags fmtflg)
    Sets **fmtflg**&format flag (**fmtfl**) to the format flag (**fmtfl**).
    Return value:
    Format flag (**fmtfl**) before setting

11. ios_base::fmtflags ios_base::setf(fmtflags fmtflg)
    Sets **fmtflg** to the format flag (**fmtfl**).
    Return value:
    Format flag (**fmtfl**) before setting

12. ios_base::fmtflags ios_base::setf((fmtflags fmtflg, fmtflags mask)
    Sets the **mask&fmtflg** value to the format flag (**fmtfl**).
    Return value:
    Format flag (**fmtfl**) before setting.

13. void ios_base::unsetf(fmtflags mask)
    Sets ~**mask**&format flag (**fmtfl**) to the format flag (**fmtfl**).

14. char ios_base::fill() const
    References the fill character (**fillch**).
    Return value:
        Fill character (**fillch**)

15. char ios_base::fill(char ch)
    Sets **ch** as the fill character.
    Return value: Fill character (**fillch**) before setting

16. int ios_base::precision() const
    References the precision (**prec**).
    Return value:
        Precision (**prec**)

17. streamsize ios_base::precision(streamsize preci)
    Sets **preci** as the precision (**prec**).
    Return value:
        Precision (**prec**) before setting

18. streamsize ios_base::width() const
    References the field width (**wide**).
    Return value:
        Field width (**wide**)

19. streamsize ios_base::width(streamsize wd)
    Sets **wd** as the field width (**wide**).
    Return value:
        Field width (**wide**) before setting

(c) ios Class

Type	Definition Name	Description
Variable	sb	Pointer to the **streambuf** object
	tiestr	Pointer to the **ostream** object
	state	State flag of **streambuf**
Function	ios()	Constructor
	ios(streambuf _far * sbptr)	
	void init(streambuf _far * sbptr)	Performs initial setting
	virtual ~ios()	Destructor
	operator void _far *() const	Tests whether an error has been generated (!**state**&(**badbit** | **failbit**)
	bool operator!() const	Tests whether an error has been generated (**state**&(**badbit** | **failbit**))
	iostate rdstate() const	References the state flag (**state**)
	void clear(iostate st = goodbit)	Clears the state flag (**state**) except for the specified state (**st**)
	void setstate(iostate st)	Specifies **st** as the state flag (**state**)
	bool good() const	Tests whether an error has been generated (**state**==**goodbit**)
	bool eof() const	Tests for the end of an input stream (**state**&**eofbit**)
	bool bad() const	Tests whether an error has been generated (**state**&**badbit**)
	bool fail() const	Tests whether the input text matches the requested pattern (**state**&(**badbit** | **failbit**))
	ostream _far * tie() const	References the pointer to the **ostream** object (**tiestr**)
	ostream _far * tie(ostream _far * tstrptr)	Sets **tstrptr** as the pointer to the **ostream** object (**tiestr**)
	streambuf _far * rdbuf() const	References the pointer to the **streambuf** object (**sb**)
	streambuf _far * rdbuf(streambuf _far * sbptr)	Sets **sbptr** as the pointer to the **streambuf** object (**sb**)
	ios _far & copyfmt (const ios _far & rhs)	Copies the state flag (**state**) of **rhs**

1. ios::ios()
   Constructor of class **ios**.
   Calls **init(0)** and sets the initial value to the member object.

2. ios::ios(streambuf _far * sbptr)
   Constructor of class **ios**.
   Calls **init(sbptr)** and sets the initial value to the member object.

3. void ios::init(streambuf _far * sbptr)
   Sets **sbptr** to **sb**.
   Sets **state** and **tiestr** to 0.

4. virtual ios::~ios()
   Destructor of class **ios**.

5. ios::operator void _far *() const
   Tests whether an error has been generated (**!state&(badbit | failbit)**).
   Return value:
   An error has been generated:
   false
   No error has been generated:
   true

6. bool ios::operator!() const
   Tests whether an error has been generated (**state&(badbit | failbit)**).
   Return value:
   An error has been generated:
   true
   No error has been generated:
   false

7. iostate ios::rdstate() const
   References the state flag (**state**).
   Return value:
   State flag (**state**)

8. void ios::clear(iostate st = goodbit)
   Clears the state flag (**state**) except for the specified state (**st**).
   If the pointer to the **streambuf** object (**sb**) is 0, **badbit** is set to the state flag (**state**).

9. void ios::setstate(iostate st)
   Sets st to the state flag (**state**).

10. bool ios::good() const
    Tests whether an error has been generated (**state==goodbit**).
    Return value:
    An error has been generated:
    false
    No error has been generated:
    true

11. bool ios::eof() const
    Tests for the end of the input stream (**state&eofbit**).
    Return value:
    End of the input stream has been reached:
    true
    End of the input stream has not been reached:
    false

12. bool ios::bad() const
    Tests whether an error has been generated (**state&badbit**).
    Return value:
    An error has been generated:
    true
    No error has been generated:
    false

13. bool ios::fail() const
    Tests whether the input text matches the requested pattern (**state&(badbit | failbit)**).
    Return value:
        Does not match the requested pattern:
            true
        Matches the requested pattern:
            false

14. ostream _far * ios::tie() const
    References the pointer (**tiestr**) to the **ostream** object.
    Return value:
        Pointer to the **ostream** object (**tiestr**)

15. ostream _far * ios::tie(ostream _far * tstrptr)
    Sets **tstrptr** as the pointer (**tiestr**) to the **ostream** object.
    Return value:
        Pointer to the **ostream** object (**tiestr**) before setting

16. streambuf _far * ios::rdbuf() const
    References the pointer to the **streambuf** object (**sb**).
    Return value:
        Pointer to the **streambuf** object (**sb**)

17. streambuf _far * ios::rdbuf(streambuf _far * sbptr)
    Sets **sbptr** as the pointer to the **streambuf** object (**sb**).
    Return value:
        Pointer to the **streambuf** object (**sb**) before setting

18. ios _far & copyfmt (const ios _far & rhs)
    Copies the state flag (**state**) of **rhs**.
    Return value:
        ***this**

(d) ios Class Manipulators

Type	Definition Name	Description
Function	ios_base _far & showbase(ios_base _far & str)	Specifies the radix display prefix mode
	ios_base _far & noshowbase(ios_base _far & str)	Clears the radix display prefix mode
	ios_base _far & showpoint (ios_base _far & str)	Specifies the decimal-point generation mode
	ios_base _far & noshowpoint (ios_base _far & str)	Clears the decimal-point generation mode
	ios_base _far & showpos(ios_base _far & str)	Specifies the + sign generation mode
	ios_base _far & noshowpos(ios_base _far & str)	Clears the + sign generation mode
	ios_base _far & skipws(ios_base _far & str)	Specifies the space skipping mode
	ios_base _far & noskipws(ios_base _far & str)	Clears the space skipping mode
	ios_base _far & uppercase(ios_base _far & str)	Specifies the uppercase letter conversion mode
	ios_base _far & nouppercase(ios_base _far & str)	Clears the uppercase letter conversion mode
	ios_base _far & internal(ios_base _far & str)	Specifies the internal fill mode
	ios_base _far & left(ios_base _far & str)	Specifies the left side fill mode
	ios_base _far & right(ios_base _far & str)	Specifies the right side fill mode
	ios_base _far & dec(ios_base _far & str)	Specifies the decimal mode
	ios_base _far & hex(ios_base _far & str)	Specifies the hexadecimal mode
	ios_base _far & oct(ios_base _far & str)	Specifies the octal mode
	ios_base _far & fixed(ios_base _far & str)	Specifies the fixed-point mode
	ios_base _far & scientific(ios_base _far & str)	Specifies the scientific description mode
	ios_base _far & boolalpha (ios_base _far & str)	Makes output of a bool-type value true or false. The return value is str.
	ios_base _far & noboolalpha (ios_base _far & str)	Sets output of a bool-type value to 1 or 0. The return value is str.

1. ios_base _far & showbase(ios_base _far & str)
   Specifies an output mode of prefixing a radix at the beginning of data.
   For a hexadecimal, 0x is prefixed. For a decimal, nothing is prefixed. For an octal, 0 is prefixed.
   Return value:
   str

2. ios_base _far & noshowbase(ios_base _far & str)
   Clears the output mode of prefixing a radix at the beginning of data.
   Return value:
   str

3. ios_base _far & showpoint(ios_base _far & str)
   Specifies the output mode of showing the decimal point.
   If no precision is specified, six decimal-point (fraction) digits are displayed.
   Return value:
   str

4. ios_base _far & noshowpoint(ios_base _far & str)
Clears the output mode of showing the decimal point.
Return value:
str

5. ios_base _far & showpos(ios_base _far & str)
Specifies the output mode of generating the + sign (adds a + sign to a positive number).
Return value:
str

6. ios_base _far & noshowpos(ios_base _far & str)
Clears the output mode of generating the + sign.
Return value:
str

7. ios_base _far & skipws(ios_base _far & str)
Specifies the input mode of skipping spaces (skips consecutive spaces).
Return value:
str

8. ios_base _far & noskipws(ios_base _far & str)
Clears the input mode of skipping spaces.
Return value:
str

9. ios_base _far & uppercase(ios_base _far & str)
Specifies the output mode of converting letters to uppercases.
In hexadecimal, the radix will be uppercase letters 0X, and the numeric value letters will be uppercase letters.
The exponential representation of a floating-point value will also use uppercase letter E.
Return value:
str

10. ios_base _far & nouppercase(ios_base _far & str)
Clears the output mode of converting letters to uppercases.
Return value:
str

11. ios_base _far & internal(ios_base _far & str)
When data is output in the field width (wide) range, it is output in the order of
Sign and radix
Fill character (fill)
Numeric value
Return value:
str

12. ios_base _far & left(ios_base _far & str)
When data is output in the field width (wide) range, it is aligned to the left.
Return value:
str

13. ios_base _far & right(ios_base _far & str)
    When data is output in the field width (**wide**) range, it is aligned to the right.
    Return value:
        str

14. ios_base _far & dec(ios_base _far & str)
    Specifies the conversion radix to the decimal mode.
    Return value:
        str

15. ios_base _far & hex(ios_base _far & str)
    Specifies the conversion radix to the hexadecimal mode.
    Return value:
        str

16. ios_base _far & oct(ios_base _far & str)
    Specifies the conversion radix to the octal mode.
    Return value:
        str

17. ios_base _far & fixed(ios_base _far & str)
    Specifies the fixed-point output mode.
    Return value:
        str

18. ios_base _far & scientific(ios_base _far & str)
    Specifies the scientific description output mode (exponential description).
    Return value:
        str

19. ios_base _far & boolalpha (ios_base _far & str)
    Makes output of a bool-type value true or false.
    Return value:
        str

20. ios_base _far & noboolalpha (ios_base _far & str)
    Sets output of a bool-type value to 1 or 0.
    Return value:
        str

(e) streambuf Class

Type	Definition Name	Description
Constant	eof	Indicates the end of the file
Variable	_B_cnt_ptr	Pointer to the length of valid data in the buffer
	B_beg_ptr	Pointer to the base pointer of the buffer
	_B_len_ptr	Pointer to the length of the buffer
	B_next_ptr	Pointer to the next position of the buffer from which data is to be read
	B_end_ptr	Pointer to the end position of the buffer
	B_beg_pptr	Pointer to the start position of the control buffer
	B_next_pptr	Pointer to the next position of the buffer from which data is to be read
	C_flg_ptr	Pointer to the input/output control flag of the file
Function	char _far * _ec2p_getflag() const	References the pointer for the file input/output control flag
	char _far * _far & _ec2p_gnptr()	References the pointer to the next position of the buffer from which data is to be read
	char _far * _far & _ec2p_pnptr()	References the pointer to the next position of the buffer where data is to be written
	void _ec2p_bcntplus()	Increments the valid data length of the buffer
	void _ec2p_bcntminus()	Decrements the valid data length of the buffer
	void _ec2p_setbPtr(char _far * _far * begptr, char _far * _far * curptr, long _far * cntptr, ong _far * lenptr, char _far * flgptr)	Sets the pointers of **streambuf**
	streambuf()	Constructor
	virtual ~streambuf()	Destructor
	streambuf _far * pubsetbuf(char _far * s, streamsize n)	Allocates the buffer for stream input/output. This function calls **setbuf (s,n)***1.
	pos_type pubseekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out)	Moves the position to read or write data in the input/output stream by using the method specified by **way**. This function calls **seekoff(off,way,which)***1.
	pos_type pubseekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out)	Calculates the offset from the beginning of the stream to the current position. This function calls **seekpos(sp,which)***1.
	int pubsync()	Flushes the output stream. This function calls **sync()***1.
	streamsize in_avail()	Calculates the offset from the end of the input stream to the current position

Type	Definition Name	Description
Function	int_type snextc()	Reads the next character
	int_type sbumpc()	Reads one character and sets the pointer to the next character
	int_type sgetc()	Reads one character
	int sgetn(char _far * s, streamsize n)	Reads **n** characters and sets them in the memory area specified by **s**
	int_type sputbackc(char c)	Puts back the read position
	int sungetc()	Puts back the read position
	int sputc(char c)	Inserts character **c**
	int_type sputn(const char _far * s, streamsize n)	Inserts **n** characters at the position pointed to by the amount specified by **s**
	char _far * eback() const	Reads the start pointer of the input stream
	char _far * gptr() const	Reads the next pointer of the input stream
	char _far * egptr() const	Reads the end pointer of the input stream
	void gbump(int n)	Moves the next pointer of the input stream by the amount specified by **n**
	void setg(char _far * gbeg, char _far * gnext, char _far * gend)	Assigns each pointer of the input stream
	char _far * pbase() const	Calculates the start pointer of the output stream
	char _far * pptr() const	Calculates the next pointer of the output stream
	char _far * epptr() const	Calculates the end pointer of the output stream
	void pbump(int n)	Moves the next pointer of the output stream by the amount specified by **n**
	void setp(char _far * pbeg, char _far * pend)	Assigns each pointer of the output stream
	virtual streambuf _far * setbuf(char _far * s, streamsize n)*[1]	For each derived class, a defined operation is executed
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))*[1]	Changes the stream position
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))*[1]	Changes the stream position
	virtual int sync()*[1]	Flushes the output stream
	virtual int showmanyc()*[1]	Calculates the number of valid characters in the input stream
	virtual streamsize xsgetn(char _far * s, streamsize n)	Sets **n** characters in the memory area specified by **s**
	virtual int_type underflow()*[1]	Reads one character without moving the stream position
	virtual int_type uflow()*[1]	Reads one character of the next pointer
	virtual int_type pbackfail(int type c = eof)*[1]	Puts back the character specified by **c**

Type	Definition Name	Description
Function	virtual streamsize xsputn(const char _far * s,streamsize n)	Inserts **n** characters in the position specified by **s**
	virtual int_type overflow(int type c = eof)*1	Inserts character **c** in the output stream

Note: *1.This class does not define the processing.

1. streambuf::streambuf()
   Constructor.
   The initial settings are as follows:
   _B_cnt_ptr = B_beg_ptr = B_next_ptr = B_end_ptr = C_flg_ptr = _B_len_ptr = 0
   B_beg_pptr = &B_beg_ptr
   B_next_pptr = &B_next_ptr

2. virtual streambuf::~streambuf()
   Destructor.

3. streambuf _far * streambuf::pubsetbuf(char _far * s, streamsize n)
   Allocates the buffer for stream input/output.
   This function calls **setbuf (s,n)**.
   Return value:
   ***this**

4. pos_type streambuf::pubseekoff(off_type off, ios_base::seekdir way,
   ios_base::openmode which = (ios_base::openmode)(ios_base::in | ios_base::out))
   Moves the read or write position for the input/output stream by using the method specified by **way**.
   This function calls **seekoff(off,way,which)**.
   Return value:
   The stream position newly specified

5. pos_type streambuf::pubseekpos(pos_type sp, ios_base::openmode which =
   (ios_base::openmode)(ios_base::in | ios_base::out))
   Calculates the offset from the beginning of the stream to the current position.
   Moves the current stream pointer by the amount specified by **sp**.
   This function calls **seekpos(sp,which)**.
   Return value:
   The offset from the beginning of the stream

6. int streambuf::pubsync()
   Flushes the output stream.
   This function calls **sync()**.
   Return value:
   0

7. streamsize streambuf::in_avail()
   Calculates the offset from the end of the input stream to the current position.
   Return value:
   If the position where data is read is valid:
   The offset from the end of the stream to the current position
   If the position where data is read is invalid:
   0 (**showmanyc()** is called)

8. int_type streambuf::snextc()
   Reads one character. If the character read is not **eof**, the next character is read.
   Return value:
      If the character read is not **eof**:
         The character read
      If the character read is **eof**:
         eof

9. int_type streambuf::sbumpc()
   Reads one character and moves forward the pointer to the next.
   Return value:
      If the position where data is read is valid:
         The character read
      If the position where data is read is invalid:
         eof

10. int_type streambuf::sgetc()
    Reads one character.
    Return value:
       If the position where data is read is valid:
          The character read
       If the position where data is read is invalid:
          eof

11. int streambuf::sgetn(char _far * s, streamsize n)
    Sets **n** characters in the memory area specified by **s**.
    If an **eof** is found in the string read, setting is stopped.
    Return value:
       The specified number of characters

12. int_type streambuf::sputbackc(char c)
    If the data read position is correct and the put back data of the position is the same as **c**, the read position is put back.
    Return value:
       If the read position was put back:
          The value of **c**
       If the read position was not put back:
          eof

13. int streambuf::sungetc()
    If the data read position is correct, the read position is put back.
    Return value:
       If the read position was put back:
          The value that was put back
       If the read position was not put back:
          eof

14. int streambuf::sputc(char c)
    Inserts character **c**.
    Return value:
        If the write position is correct:
            The value of **c**
        If the write position is incorrect:
            Eof

15. int_type streambuf::sputn(const char _far * s, streamsize n)
    Inserts **n** characters at the position specified by **s**.
    If the buffer is smaller than **n**, the number of characters for the buffer is inserted.
    Return value:
        The number of characters inserted

16. char _far * streambuf::eback() const
    Calculates the start pointer of the input stream.
    Return value:
        Start pointer

17. char _far * streambuf::gptr() const
    Calculates the next pointer of the input stream.
    Return value:
        Next pointer

18. char _far * streambuf::egptr() const
    Calculates the end pointer of the input stream.
    Return value:
        End pointer

19. void streambuf::gbump(int n)
    Moves forward the next pointer of the input stream by the amount specified by **n**.

20. void streambuf::setg(char _far * gbeg, char _far * gnext, char _far * gend)
    Sets each pointer of the input stream as follows:
        *B_beg_pptr = gbeg;
        *B_next_pptr = gnext;
        B_end_ptr = gend;
        *_B_cnt_ptr = gend-gnext;
        *_B_len_ptr = gend-gbeg;

21. char _far * streambuf::pbase() const
    Calculates the start pointer of the output stream.
    Return value:
        Start pointer

22. char _far * streambuf::pptr() const
    Calculates the next pointer of the output stream.
    Return value:
        Next pointer

23. char _far * streambuf::epptr() const
    Calculates the end pointer of the output stream.
    Return value:
        End pointer

24. void streambuf::pbump(int n)
    Moves forward the next pointer of the output stream by the amount specified by **n**.

25. void streambuf::setp(char _far * pbeg, char _far * pend)
    The settings for each pointer of the output stream are as follows:
        *B_beg_pptr = pbeg;
        *B_next_pptr = pbeg;
        B_end_ptr = pend;
        *_B_cnt_ptr = pend-pbeg;
        *_B_len_ptr = pend-pbeg;

26. virtual streambuf _far * streambuf::setbuf(char _far * s, streamsize n)
    For each derived class from **streambuf**, a defined operation is executed.
    Return value:
        ***this** (This class does not define the processing.)

27. virtual pos_type streambuf::seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode)(ios_base::in | ios_base::out))
    Changes the stream position.
    Return value:
        -1 (This class does not define the processing.)

28. virtual pos_type streambuf::seekpos(pos_type sp, ios_base::openmode =(ios_base::openmode) (ios_base::in | ios_base::out))
    Changes the stream position.
    Return value:
        -1 (This class does not define the processing.)

29. virtual int streambuf::sync()
    Flushes the output stream.
    Return value:
        0 (This class does not define the processing.)

30. virtual int streambuf::showmanyc()
    Calculates the number of valid characters in the input stream.
    Return value:
        0 (This class does not define the processing.)

31. virtual streamsize streambuf::xsgetn(char _far * s, streamsize n)
    Sets **n** characters in the memory area specified by **s**.
    If the buffer is smaller than **n**, the number of characters for the buffer is inserted.
    Return value:
        The number of characters input

32. virtual int_type streambuf::underflow()
Reads one character without moving the stream position.
Return value:
eof (This class does not define the processing.)

33. virtual int_type streambuf::uflow()
Reads one character of the next pointer.
Return value:
eof (This class does not define the processing.)

34. virtual int_type streambuf::pbackfail(int_type c = eof)
Puts back the character specified by **c**.
Return value:
eof (This class does not define the processing.)

35. virtual streamsize streambuf::xsputn(const char _far * s, streamsize n)
Inserts **n** characters specified by **s** in to the stream position.
If the buffer is smaller than **n**, the number of characters for the buffer is inserted.
Return value:
The number of characters inserted

36. virtual int_type streambuf::overflow(int_type c = eof)
Inserts character **c** in the output stream.
Return value:
eof (This class does not define the processing.)

(f) istream::sentry Class

Type	Definition Name	Description
Variable	ok_	Whether the current state is input-enabled
Function	sentry(istream _far & is, bool noskipws = false)	Constructor
	~sentry()	Destructor
	operator bool()	References **ok_**

1. istream::sentry::sentry(istream _far & is, bool noskipws = _false)
   Constructor of internal class **sentry**.
   If **good()** is non-zero, enables input with or without a format.
   If **tie()** is non-zero, flushes the related output stream.

2. istream::sentry::~sentry()
   Destructor of internal class **sentry**.

3. istream::sentry::operator bool()
   References **ok_**.
   Return value:
   ok_

(g) istream Class

Type	Definition Name	Description
Variable	chcount	The number of characters extracted by the input function called last
Function	int _ec2p_getistr(char _far * str, unsigned int dig, int mode)	Converts **str** with the radix specified by **dig**
	istream(streambuf _far * sb)	Constructor
	virtual ~istream()	Destructor
	istream _far & operator>>(bool _far & n)	Stores the extracted characters in **n**
	istream _far & operator>>(short _far & n)	
	istream _far & operator>>(unsigned short _far & n)	
	istream _far & operator>>(int _far & n)	
	istream _far & operator>>(unsigned int _far & n)	
	istream _far & operator>>(long _far & n)	
	istream _far & operator>>(unsigned long _far & n)	
	istream _far & operator>>(long long _far & n)	
	istream _far & operator>>(unsigned long long _far & n)	
	istream _far & operator>>(float _far & n)	
	istream _far & operator>>(double _far & n)	
	istream _far & operator>>(long double _far & n)	
	istream _far & operator>>(void _far * _far & p)	Converts the extracted characters to a pointer to **void** and stores them in **p**
	istream _far & operator>>(streambuf _far * sb)	Extracts characters and stores them in the memory area specified by **sb**
	streamsize gcount() const	Calculates **chcount** (number of characters extracted)
	int_type get()	Extracts a character
	istream _far & get(char _far & c)	Extracts characters and stores them in **c**
	istream _far & get(signed char _far & c)	
	istream _far & get(unsigned char _far & c)	
	istream _far & get(char _far * s, streamsize n)	Extracts strings with size **n-1** and stores them in the memory area specified by **s**
	istream _far & get(signed char _far * s, streamsize n)	
	istream _far & get(unsigned char _far * s, streamsize n)	
	istream _far & get(char _far * s, streamsize n, char delim)	Extracts strings with size **n-1** and stores them in the memory area specified by **s**. If **delim** is found in the string, input is stopped.
	istream _far & get(signed char _far * s, streamsize n, char delim)	
	istream _far & get(unsigned char _far * s, streamsize n, char delim)	

Type	Definition Name	Description
Function	istream _far & get(streambuf _far & sb)	Extracts strings and stores them in the memory area specified by **sb**
	istream _far & get(streambuf _far & sb, char delim)	Extracts strings and stores them in the memory area specified by **sb**. If **delim** is found in the string, input is stopped.
	istream _far & getline(char _far * s, streamsize n)	Extracts strings with size **n-1** and stores them in the memory area specified by **s**.
	istream _far & getline(signed char _far * s, streamsize n)	
	istream _far & getline(unsigned char _far * s, streamsize n)	
	istream _far & getline(char _far * s, streamsize n, char delim)	Extracts strings with size **n-1** and stores them in the memory area specified by **s**. If **delim** is found in the string, input is stopped.
	istream _far & getline(signed char _far * s, streamsize n, char delim)	
	istream _far & getline(unsigned char _far * s, streamsize n, char delim)	
	istream _far & ignore(streamsize n = 1, int_type delim = streambuf::eof)	Skips reading the number of characters specified by **n**. If **delim** is found in the string, skipping is stopped.
	int_type peek()	Seeks for input characters that can be acquired next
	istream _far & read(char _far * s, streamsize n)	Extracts strings with size **n** and stores them in the memory area specified by **s**
	istream _far & read(signed char _far * s, streamsize n)	
	istream _far & read(unsigned char _far * s, streamsize n)	
	streamsize readsome(char _far * s, streamsize n)	Extracts strings with size **n** and stores them in the memory area specified by **s**
	streamsize readsome(signed char _far * s, streamsize n)	
	streamsize readsome(unsigned char _far * s, streamsize n)	
	istream _far & putback(char c)	Puts back a character to the input stream.
	istream _far & unget()	Puts back the position of the input stream.
	int sync()	Checks the existence of the input stream. This function calls **streambuf::pubsync()**.
	pos_type tellg()	Finds the input stream position. This function calls **streambuf::pubseekoff(0,cur,in)**.
	istream _far & seekg(pos_type pos)	Moves the current stream pointer by the amount specified by **pos**. This function calls **streambuf::pubseekpos(pos)**.
	istream _far & seekg(off_type off, ios_base::seekdir dir)	Moves the position to read the input stream by using the method specified by **dir**. This function calls **streambuf::pubseekoff(off,dir)**.

1. int istream::_ec2p_getistr(char _far * str, unsigned int dig, int mode)
   Converts **str** to the radix specified by **dig**.
   Return value:
   The converted radix

2. istream::istream(streambuf _far * sb)
   Constructor of class **istream**.
   Calls **ios::init(sb)**.
   Specifies **chcount**=0.

3. virtual istream::~istream()
   Destructor of class **istream**.

4. istream _far & istream::operator>>(bool _far & n)
   istream _far & istream::operator>>(short _far & n)
   istream _far & istream::operator>>(unsigned short _far & n)
   istream _far & istream::operator>>(int _far & n)
   istream _far & istream::operator>>(unsigned int _far & n)
   istream _far & istream::operator>>(long _far & n)
   istream _far & istream::operator>>(unsigned long _far & n)
   istream _far & istream::operator>>(long long _far & n)
   istream _far & istream::operator>>(unsigned long long _far & n)
   istream _far & istream::operator>>(float _far & n)
   istream _far & istream::operator>>(double _far & n)
   istream _far & istream::operator>>(long double _far & n)
   Stores the extracted characters in **n**.
   Return value:
   ***this**

5. istream _far & istream::operator>>(void _far * _far & p)
   Converts the extracted characters to a **void*** type and stores them in the memory specified by **p**.
   Return value:
   ***this**

6. istream _far & istream::operator>>(streambuf _far * sb)
   Extracts characters and stores them in the memory area specified by **sb**.
   If there are no extracted characters, **setstate(failbit)** is called.
   Return value:
   ***this**

7. streamsize istream::gcount() const
   References **chcount** (number of extracted characters).
   Return value:
   chcount

8. int_type istream::get()
   Extracts characters.
   Return value:
      If characters are extracted:
         Extracted characters.
      If no characters are extracted:
         Calls **setstate(failbit)** and becomes **streambuf::eof**.

9. istream _far & istream::get(char _far & c)
   istream _far & istream::get(signed char _far & c)
   istream _far & istream::get(unsigned char _far & c)
   Extracts characters and stores them in **c**.
   If the extracted character is **streambuf::eof**, **failbit** is set.
   Return value:
      ***this**

10. istream _far & istream::get(char _far * s, streamsize n)
    istream _far & istream::get(signed char _far * s, streamsize n)
    istream _far & istream::get(unsigned char _far * s, streamsize n)
    Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
    If **ok_==false** or no character has been extracted, **failbit** is set.
    Return value:
       ***this**

11. istream _far & istream::get(char _far * s, streamsize n, char delim)
    istream _far & istream::get(signed char _far * s, streamsize n, char delim)
    istream _far & istream::get(unsigned char _far * s, streamsize n, char delim)
    Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
    If **delim** is found in the string, input is stopped.
    If **ok_==false** or no character has been extracted, **failbit** is set.
    Return value:
       ***this**

12. istream _far & istream::get(streambuf _far & sb)
    Extracts a string and stores it in the memory area specified by **sb**.
    If **ok_==false** or no character has been extracted, **failbit** is set.
    Return value:
       ***this**

13. istream _far & istream::get(streambuf _far & sb, char delim)
    Extracts a string and stores it in the memory area specified by **sb**.
       If **delim** is found in the string, input is stopped.
       If **ok_==false** or no character has been extracted, **failbit** is set.
    Return value:
       ***this**

14. istream _far & istream::getline(char _far * s, streamsize n)
istream _far & istream::getline(signed char _far * s, streamsize n)
istream _far & istream::getline(unsigned char _far * s, streamsize n)
Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If **ok_==false** or no character has been extracted, **failbit** is set.
Return value:
***this**

15. istream _far & istream::getline(char _far * s, streamsize n, char delim)
istream _far & istream::getline(signed char _far * s, streamsize n, char delim)
istream _far & istream::getline(unsigned char _far * s, streamsize n, char delim)
Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If character **delim** is found, input is stopped.
If **ok_==false** or no character has been extracted, **failbit** is set.
Return value:
***this**

16. istream _far & istream::ignore(streamsize n = 1, int_type delim = streambuf::eof)
Skips reading the number of characters specified by **n**.
If character **delim** is found, skipping is stopped.
Return value:
***this**

17. int_type istream::peek()
Seeks input characters that will be available next.
Return value:
If **ok_==false**:
streambuf::eof
If **ok_!=false**:
rdbuf()->sgetc()

18. istream _far & istream::read(char _far * s, streamsize n)
istream _far & istream::read(signed char _far * s, streamsize n)
istream _far & istream::read(unsigned char _far * s, streamsize n)
If **ok_!=false**, extracts a string with size **n** and stores it in the memory area specified by **s**.
If the number of extracted characters does not match with the number of **n**, **eofbit** is set.
Return value:
***this**

19. streamsize istream::readsome(char _far * s, streamsize n)
streamsize istream::readsome(signed char _far * s, streamsize n)
streamsize istream::readsome(unsigned char _far * s, streamsize n)
Extracts a string with size **n** and stores it in the memory area specified by **s**.
If the number of characters exceeds the stream size, only the number of characters equal to the stream size is stored.
Return value:
The number of extracted characters

20. istream _far & istream::putback(char c)
Puts back character **c** to the input stream.
If the characters put back are **streambuf::eof**, **badbit** is set.
Return value:
***this**

21. istream _far & istream::unget()
Puts back the pointer of the input stream by one.
If the extracted characters are **streambuf::eof**, **badbit** is set.
Return value:
***this**

22. int istream::sync()
Checks for an input stream.
This function calls **streambuf::pubsync()**.
Return value:
If there is no input stream:
streambuf::eof
If there is an input stream:
0

23. pos_type istream::tellg()
Checks for the position of the input stream.
This function calls **streambuf::pubseekoff(0,cur,in)**.
Return value:
Offset from the beginning of the stream
If an error occurs during the input processing, -1 is returned.

24. istream _far & istream::seekg(pos_type pos)
Moves the current stream pointer by the amount specified by **pos**.
This function calls **streambuf::pubseekpos(pos)**.
Return value:
***this**

25. istream _far & istream::seekg(off_type off, ios_base::seekdir dir)
Moves the position to read the input stream using the method specified by **dir**.
This function calls **streambuf::pubseekoff(off,dir)**.
If an error occurs during the input processing, this processing is not performed.
Return value:
***this**

(h) istream Class Manipulator

Type	Definition Name	Description
Function	istream _far & ws(istream _far & is)	Skips reading the spaces

1. istream _far & ws(istream _far & is)
   Skips reading white spaces.
   Return value:
   is

(i) istream Non-Member Function

Type	Definition Name	Description
Function	istream _far & operator>>(istream _far & in, char _far * s)	Extracts a string and stores it in the memory area specified by **s**
	istream _far & operator>>(istream _far & in, signed char _far * s)	
	istream _far & operator>>(istream _far & in, unsigned char _far * s)	
	istream _far & operator>>(istream _far & in, char _far & c)	Extracts a character and stores it in **c**
	istream _far & operator>>(istream _far & in, singed char _far & c)	
	istream _far & operator>>(istream _far & in, unsigned char _far & c)	

1. istream _far & operator>>(istream _far & in, char _far * s)
   istream _far & operator>>(istream _far & in, signed char _far * s)
   istream _far & operator>>(istream _far & in, unsigned char _far * s)
   Extracts a string and stores it in the memory area specified by **s**.
   Processing is stopped if the number of characters stored is equal to field width – 1 **streambuf::eof** is found in the input stream the next available character **c** satisfies **isspace(c)==1**
   If no characters are stored, **failbit** is set.
   Return value:
   in

2. istream _far & operator>>(istream _far & in, char _far & c)
   istream _far & operator>>(istream _far & in, singed char _far & c)
   istream _far & operator>>(istream _far & in, unsigned char _far & c)
   Extracts a character and stores it in **c**. If no character is stored, **failbit** is set.
   Return value:
   in

(j) ostream::sentry Class

Type	Definition Name	Description
Variable	ok_	Whether or not the current state allows output
	_ _ec2p_os	Pointer to the **ostream** object
Function	sentry(ostream _far & os)	Constructor
	~sentry()	Destructor
	operator bool()	References **ok_**

1. ostream::sentry::sentry(ostream _far & os)
   Constructor of the internal class **sentry**.
   If **good()** is non-zero and **tie()** is non-zero, **flush()** is called.
   Specifies **os** to **_ _ec2p_os**.

2. ostream::sentry::~sentry()
   Destructor of internal class **sentry**.
   If (**_ _ec2p_os->flags() & ios_base::unitbuf**) is true, **flush()** is called.

3. ostream::sentry::operator bool()
   References **ok_**.
   Return value:

   ok_

(k) ostream Class

Type	Definition Name	Description
Function	ostream(streambuf _far * sbptr)	Constructor.
	virtual ~ostream()	Destructor.
	ostream _far & operator<<(bool n)	Inserts **n** in the output stream.
	ostream _far & operator<<(short n)	
	ostream _far & operator<<(unsigned short n)	
	ostream _far & operator<<(int n)	
	ostream _far & operator<<(unsigned int n)	
	ostream _far & operator<<(long n)	
	ostream _far & operator<<(unsigned long n)	
	ostream _far & operator<<(long long n)	
	ostream _far & operator<<(unsigned long long n)	
	ostream _far & operator<<(float n)	
	ostream _far & operator<<(double n)	
	ostream _far & operator<<(long double n)	
	ostream _far & operator<<(void _far * n)	
	ostream _far & operator<<(streambuf _far * sbptr)	Inserts the output string of **sbptr** into the output stream.
	ostream _far & put(char c)	Inserts character **c** into the output stream.
	ostream _far & write(const char _far * s, streamsize n)	Inserts **n** characters from **s** into the output stream.
	ostream _far & write(const signed char _far * s, streamsize n)	
	ostream _far & write(const unsigned char _far * s, streamsize n)	
	ostream _far & flush()	Flushes the output stream. This function calls **streambuf::pubsync()**.
	pos_type tellp()	Calculates the current write position. This function calls **streambuf::pubseekoff(0,cur,out)**.
	ostream _far & seekp(pos_type pos)	Calculates the offset from the beginning of the stream to the current position. Moves the current stream pointer by the amount specified by **pos**. This function calls **streambuf::pubseekpos(pos)**.
	ostream _far & seekp(off_type off, seekdir dir)	Moves the stream write position by the amount specified by **off**, from **dir**. This function calls **streambuf::pubseekoff(off,dir)**.

1. ostream::ostream(streambuf _far * sbptr)
   Constructor.
   Calls **ios(sbptr)**.

2. virtual ostream::~ostream()
   Destructor.

3. ostream _far & ostream::operator<<(bool n)
   ostream _far & ostream::operator<<(short n)
   ostream _far & ostream::operator<<(unsigned short n)
   ostream _far & ostream::operator<<(int n)
   ostream _far & ostream::operator<<(unsigned int n)
   ostream _far & ostream::operator<<(long n)
   ostream _far & ostream::operator<<(unsigned long n)
   ostream _far & ostream::operator<<(long long n)
   ostream _far & ostream::operator<<(unsigned long long n)
   ostream _far & ostream::operator<<(float n)
   ostream _far & ostream::operator<<(double n)
   ostream _far & ostream::operator<<(long double n)
   ostream _far & ostream::operator<<(void _far * n)
   If **sentry::ok_==true**, **n** is inserted into the output stream.
   If **sentry::ok_==false**, **failbit** is set.
   Return value:
   ***this**

4. ostream _far & ostream::operator<<(streambuf _far * sbptr)
   If **sentry::ok_==true**, the output string of **sbptr** is inserted into the output stream.
   If **sentry::ok_==false**, **failbit** is set.
   Return value:
   ***this**

5. ostream _far & ostream::put(char c)
   If (**sentry::ok_==true**) and (**rdbuf()->sputc(c)!=streambuf::eof**), **c** is inserted into the output stream.
   Otherwise **badbit** is set.
   Return value:
   ***this**

6. ostream _far & ostream::write(const char _far * s, streamsize n)
   ostream _far & ostream::write(const signed char _far * s, streamsize n)
   ostream _far & ostream::write(const unsigned char _far * s, streamsize n)
   If (**sentry::ok_==true**) and (**rdbuf()->sputn(s, n)==n**), **n** characters specified by **s** are inserted into the output stream.
   Otherwise **badbit** is set.
   Return value:
   ***this**

7. ostream _far & ostream::flush()
   Flushes the output stream.
   This function calls **streambuf::pubsync()**.
   Return value:
   ***this**

8. pos_type ostream::tellp()
Calculates the current write position.
This function calls **streambuf::pubseekoff(0,cur,out)**.
Return value:
The current stream position
If an error occurs during processing, -1 is returned.

9. ostream _far & ostream::seekp(pos_type pos)
If no error occurs, the offset from the beginning of the stream to the current position is calculated.
Moves the current stream pointer by the amount specified by **pos**.
This function calls **streambuf::pubseekpos(pos)**.
Return value:
***this**

10. ostream _far & ostream::seekp(off_type off, seekdir dir)
If no error occurs, the stream write position is moved by the amount specified by **off**, from **dir**.
This function calls **streambuf::pubseekoff(off,dir)**.
Return value:*
this

(l) ostream Class Manipulator

Type	Definition Name	Description
Function	ostream _far & endl(ostream _far & os)	Inserts a new line and flushes the output stream
	ostream _far & ends(ostream _far & os)	Inserts a **NULL** code
	ostream _far & flush(ostream _far & os)	Flushes the output stream

1. ostream _far & endl(ostream _far & os)
Inserts a new line code and flushes the output stream.
This function calls **flush()**.
Return value:
os

2. ostream _far & ends(ostream _far & os)
Inserts a **NULL** code into the output line.
Return value:
os

3. ostream _far & flush(ostream _far & os)
Flushes the output stream.
This function calls **streambuf::sync()**.
Return value:
os

(m) ostream Non-Member Function

Type	Definition Name	Description
Function	ostream _far & operator<<(ostream _far & os, char s)	Inserts **s** into the output stream
	ostream _far & operator<<(ostream _far & os, signed char s)	
	ostream _far & operator<<(ostream _far & os, unsigned char s)	
	ostream _far & operator<<(ostream _far & os, const char _far * s)	
	ostream _far & operator<<(ostream _far & os, const singed char _far * s)	
	ostream _far & operator<<(ostream _far & os, const unsigned char _far * s)	

1. ostream _far & operator<<(ostream _far & os, char s)
   ostream _far & operator<<(ostream _far & os, signed char s)
   ostream _far & operator<<(ostream _far & os, unsigned char s)
   ostream _far & operator<<(ostream _far & os, const char _far * s)
   ostream _far & operator<<(ostream _far & os, const singed char _far * s)
   ostream _far & operator<<(ostream _far & os, const unsigned char _far * s)
   If (**sentry::ok_==true**) and an error does not occur, **s** is inserted into the output stream.
   Otherwise **failbit** is set.
   Return value:
   os

(n) smanip Class Manipulator

Type	Definition Name	Description
Function	smanip resetiosflags(ios_base::fmtflags mask)	Clears the flag specified by the **mask** value
	smanip setiosflags(ios_base::fmtflags mask)	Specifies the format flag (**fmtfl**)
	smanip setbase(int base)	Specifies the radix used at output
	smanip setfill(char c)	Specifies the fill character (**fillch**)
	smanip setprecision(int n)	Specifies the precision (**prec**)
	smanip setw(int n)	Specifies the field width (**wide**)

1. smanip resetiosflags(ios_base::fmtflags mask)
   Clears the flag specified by the **mask** value.
   Return value:
   Target object of input/output

2. smanip setiosflags(ios_base::fmtflags mask)
   Specifies the format flag (**fmtfl**).
   Return value:
   Target object of input/output

3. smanip setbase(int base)
   Specifies the radix used at output.
   Return value:
   Target object of input/output

4. smanip setfill(char c)
   Specifies the fill character (**fillch**).
   Return value:
   Target object of input/output

5. smanip setprecision(int n)
   Specifies the precision (**prec**).
   Return value:
   Target object of input/output

6. smanip setw(int n)
   Specifies the field width (**wide**).
   Return value:
   Target object of input/output

(3) Memory Management Library

The header file for the memory management library is as follows:

- <new>
  Defines the memory allocation/deallocation function.

By setting an exception handling function address to the **_ec2p_new_handler** variable, exception handling can be executed if memory allocation fails. The **_ec2p_new_handler** is a **static** variable and the initial value is **NULL**. If this handler is used, reentrance will be lost.

Operations required for the exception handling function:

- Creates an allocatable area and returns the area.
- Operations are not prescribed for cases where an area cannot be created.

Type	Definition Name	Description
Type	new_handler	Pointer type to the function that returns a **void** type
Variable	_ec2p_new_handler	Pointer to an exception handling function
Function	void _far * operator new(size_t size)	Allocates a memory area with a size specified by **size**
	void _far * operator new[ ](size_t size)	Allocates an array area with a size specified by **size**
	void _far * operator new(size_t size, void _far * ptr)	Allocates the area specified by **ptr** as the memory area
	void _far * operator new[ ](size_t size, void _far * ptr)	Allocates the area specified by **ptr** as the array area
	void operator delete(void _far * ptr)	Deallocates the memory area
	void operator delete[ ](void _far * ptr)	Deallocates the array area
	new_handler set_new_handler(new_handler new_P)	Sets the exception handling function address (**new_P**) in **_ec2p_new_handler**

1. void _far * operator new(size_t size)
   Allocates a memory area with the size specified by **size**.
   If memory allocation fails and when **new_handler** is set, **new_handler** is called.
   Return value:
   If memory allocation succeeds:
   Pointer to **void** type
   If memory allocation fails:
   NULL

2. void _far * operator new[ ](size_t size)
   Allocates an array area with the size specified by **size**.
   If memory allocation fails and when **new_handler** is set, **new_handler** is called.
   Return value:
   If memory allocation succeeds:
   Pointer to **void** type
   If memory allocation fails:
   NULL

3. void _far * operator new(size_t size, void _far * ptr)
   Allocates the area specified by **ptr** as the storage area.
   Return value:
   ptr

4. void _far * operator new[ ](size_t size, void _far * ptr)
   Allocates the area specified by **ptr** as the array area.
   Return value:
   ptr

5. void operator delete(void _far * ptr)
   Deallocates the storage area specified by **ptr**.
   If **ptr** is **NULL**, no operation will be performed.

6. void operator delete[ ](void _far * ptr)
   Deallocates the array area specified by **ptr**.
   If **ptr** is **NULL**, no operation will be performed.

7. new_handler set_new_handler(new_handler new_P)
   Sets **new_P** to **_ec2p_new_handler**.
   Return value:
   _ec2p_new_handler

(4) Complex Number Calculation Class Library

The header file for the complex number calculation class library is as follows:

- <complex>
  Defines the **float_complex** and **double_complex** classes.

These classes have no derivation.

(a) float_complex Class

Type	Definition Name	Description
Type	value_type	**float** type
Variable	_re	Defines the real part of **float** precision
	_im	Defines the imaginary part of **float** precision
Function	float_complex(float re = 0.0f, float im = 0.0f)	Constructor
	float_complex(const double_complex _far & rhs)	
	float real() const	Acquires the real part (**_re**)
	float imag() const	Acquires the imaginary part (**_im**)
	float_complex _far & operator=(float rhs)	Copies **rhs** to the real part. 0.0f is assigned to the imaginary part.
	float_complex _far & operator+=(float rhs)	Adds **rhs** to the real part and stores the sum in ***this**.
	float_complex _far & operator-=(float rhs)	Subtracts **rhs** from the real part and stores the difference in ***this**.
	float_complex _far & operator*=(float rhs)	Multiplies ***this** by **rhs** and stores the product in ***this**.
	float_complex _far & operator/=(float rhs)	Divides ***this** by **rhs** and stores the quotient in ***this**.
	float_complex _far & operator=(const float_complex _far & rhs)	Copies **rhs**.
	float_complex _far & operator+=(const float_complex _far & rhs)	Adds **rhs** to ***this** and stores the sum in ***this**.
	float_complex _far & operator-=(const float_complex _far & rhs)	Subtracts **rhs** from ***this** and stores the difference in ***this**.
	float_complex _far & operator*=(const float_complex _far & rhs)	Multiplies ***this** by **rhs** and stores the product in ***this**.
	float_complex _far & operator/=(const float_complex _far & rhs)	Divides ***this** by **rhs** and stores the quotient in ***this**.

1. float_complex::float_complex(float re = 0.0f, float im = 0.0f)
   Constructor of class **float_complex**.
   The initial settings are as follows:
   _re = re;
   _im = im;

2. float_complex::float_complex(const double_complex _far & rhs)
   Constructor of class **float_complex**.
   The initial settings are as follows:
   _re = (float)rhs.real();
   _im = (float)rhs.imag();

3. float float_complex::real() const
   Acquires the real part.
   Return value:
   this->_re

4. float float_complex::imag() const
   Acquires the imaginary part.
   Return value:
   this->_im

5. float_complex _far & float_complex::operator=(float rhs)
   Copies **rhs** to the real part (**_re**).
   0.0f is assigned to the imaginary part (**_im**).
   Return value:
   ***this**

6. float_complex _far & float_complex::operator+=(float rhs)
   Adds **rhs** to the real part (**_re**) and stores the result in the real part (**_re**).
   The value of the imaginary part (**_im**) does not change.
   Return value:
   ***this**

7. float_complex _far & float_complex::operator-=(float rhs)
   Subtracts **rhs** from the real part (**_re**) and stores the result in the real part (**_re**).
   The value of the imaginary part (**_im**) does not change.
   Return value:
   ***this**

8. float_complex _far & float_complex::operator*=(float rhs)
   Multiplies ***this** by **rhs** and stores the result in ***this**.
   (_re=_re*rhs, _im=_im*rhs)
   Return value:
   ***this**

9. float_complex _far & float_complex::operator/=(float rhs)
   Divides ***this** by **rhs** and stores the result in ***this**.
   (_re=_re/rhs, _im=_im/rhs)
   Return value:
   ***this**

10. float_complex _far & float_complex::operator=(const float_complex _far & rhs)
    Copies **rhs** to ***this**.
    Return value:
    ***this**

11. float_complex _far & float_complex::operator+=(const float_complex _far & rhs)
    Adds **rhs** to ***this** and stores the result in ***this**
    Return value:
    ***this**

12. float_complex _far & float_complex::operator-=(const float_complex _far & rhs)
    Subtracts **rhs** from ***this** and stores the result in ***this**.
    Return value:
        ***this**

13. float_complex _far & float_complex::operator*=(const float_complex _far & rhs)
    Multiplies ***this** by **rhs** and stores the result in ***this**.
    Return value:
        ***this**

14. float_complex _far & float_complex::operator/=(const float_complex _far & rhs)
    Divides ***this** by **rhs** and stores the result in ***this**.
    Return value:
        ***this**

(b) float_complex Non-Member Function

Type	Definition Name	Description
Function	float_complex operator+(const float_complex _far & lhs)	Performs unary + operation of **lhs**
	float_complex operator+(const float_complex _far & lhs, const float_complex _far & rhs)	Adds **lhs** to **rhs** and stores the sum in **lhs**
	float_complex operator+(const float_complex _far & lhs, const float _far & rhs)	
	float_complex operator+(const float _far & lhs, const float_complex _far & rhs)	
	float_complex operator-(const float_complex _far & lhs)	Performs unary - operation of **lhs**
	float_complex operator-(const float_complex _far & lhs, const float_complex _far & rhs)	Subtracts **rhs** from **lhs** and stores the difference in **lhs**
	float_complex operator-(const float_complex _far & lhs, const float _far & rhs)	
	float_complex operator-(const float _far & lhs, const float_complex _far & rhs)	
	float_complex operator*(const float_complex _far & lhs, const float_complex _far & rhs)	Multiples **lhs** by **rhs** and stores the product in **lhs**
	float_complex operator*(const float_complex _far & lhs, const float _far & rhs)	
	float_complex operator*(const float _far & lhs, const float_complex _far & rhs)	
	float_complex operator/(const float_complex _far & lhs, const float_complex _far & rhs)	Divides **lhs** by **rhs** and stores the quotient in **lhs**
	float_complex operator/(const float_complex _far & lhs, const float _far & rhs)	
	float_complex operator/(const float _far & lhs, const float_complex _far & rhs)	

Type	Definition Name	Description
Function	bool operator==(const float_complex _far & lhs, const float_complex _far & rhs)	Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**
	bool operator==(const float_complex _far & lhs, const float _far & rhs)	
	bool operator==(const float _far & lhs, const float_complex _far & rhs)	
	bool operator!=(const float_complex _far & lhs, const float_complex _far & rhs)	Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**
	bool operator!=(const float_complex _far & lhs, const float _far & rhs)	
	bool operator!=(const float _far & lhs, const float_complex _far & rhs)	
	istream _far & operator>>(istream _far & is, float_complex _far & x)	Inputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part)
	ostream _far & operator<<(ostream _far & os, float_complex _far & x)	Outputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part)
	float real(const float_complex _far & x)	Acquires the real part
	float imag(const float_complex _far & x)	Acquires the imaginary part
	float abs(const float_complex _far & x)	Calculates the absolute value
	float arg(const float_complex _far & x)	Calculates the phase angle
	float norm(const float_complex _far & x)	Calculates the absolute value of the square
	float_complex conj(const float_complex _far & x)	Calculates the conjugate complex number
	float_complex polar(const float _far & rho, const float _far & theta)	Calculates the **float_complex** value for a complex number with size **rho** and phase angle **theta**
	float_complex cos(const float_complex _far & x)	Calculates the complex cosine
	float_complex cosh(const float_complex _far & x)	Calculates the complex hyperbolic cosine
	float_complex exp(const float_complex _far & x)	Calculates the exponent function
	float_complex log(const float_complex _far & x)	Calculates the natural logarithm
	float_complex log10(const float_complex _far & x)	Calculates the common logarithm

Type	Definition Name	Description
Function	float_complex pow(const float_complex _far & x, int y)	Calculates **x** to the **y**th power
	float_complex pow(const float_complex _far & x, const float _far & y)	
	float_complex pow(const float_complex _far & x, const float_complex _far & y)	
	float_complex pow(const float _far & x, const float_complex _far & y)	
	float_complex sin(const float_complex _far & x)	Calculates the complex sine
	float_complex sinh(const float_complex _far & x)	Calculates the complex hyperbolic sine
	float_complex sqrt(const float_complex _far & x)	Calculates the square root within the right half space
	float_complex tan(const float_complex _far & x)	Calculates the complex tangent
	float_complex tanh(const float_complex _far & x)	Calculates the complex hyperbolic tangent

1. float_complex operator+(const float_complex _far & lhs)
   Performs unary + operation of **lhs**.
   Return value:
   lhs

2. float_complex operator+(const float_complex _far & lhs, const float_complex _far & rhs)
   float_complex operator+(const float_complex _far & lhs, const float _far & rhs)
   float_complex operator+(const float _far & lhs, const float_complex _far & rhs)
   Adds **lhs** to **rhs** and stores the result in **lhs**.
   Return value:
   float_complex(lhs)+=rhs

3. float_complex operator-(const float_complex _far & lhs)
   Performs unary - operation of **lhs**.
   Return value:
   float_complex(-lhs.real(), -lhs.imag())

4. float_complex operator-(const float_complex _far & lhs, const float_complex _far & rhs)
   float_complex operator-(const float_complex _far & lhs, const float _far & rhs)
   float_complex operator-(const float _far & lhs, const float_complex _far & rhs)
   Subtracts **rhs** from **lhs** and stores the result in **lhs**.
   Return value:
   float_complex(lhs)-=rhs

5. float_complex operator*(const float_complex _far & lhs, const float_complex _far & rhs)
   float_complex operator*(const float_complex _far & lhs, const float _far & rhs)
   float_complex operator*(const float _far & lhs, const float_complex _far & rhs)
   Multiples **lhs** by **rhs** and stores the result in **lhs**.
   Return value:
   float_complex(lhs)*=rhs

6. float_complex operator/(const float_complex _far & lhs, const float_complex _far & rhs)
   float_complex operator/(const float_complex _far & lhs, const float _far & rhs)
   float_complex operator/(const float _far & lhs, const float_complex _far & rhs)
   Divides **lhs** by **rhs** and stores the result in **lhs**.
   Return value:
   float_complex(lhs)/=rhs

7. bool operator==(const float_complex _far & lhs, const float_complex _far & rhs)
   bool operator==(const float_complex _far & lhs, const float _far & rhs)
   bool operator==(const float _far & lhs, const float_complex _far & rhs)
   Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
   For a **float** type parameter, the imaginary part is assumed to be 0.0f.
   Return value:
   lhs.real()==rhs.real() && lhs.imag()==rhs.imag()

8. bool operator!=(const float_complex _far & lhs, const float_complex _far & rhs)
   bool operator!=(const float_complex _far & lhs, const float _far & rhs)
   bool operator!=(const float _far & lhs, const float_complex _far & rhs)
   Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
   For a **float** type parameter, the imaginary part is assumed to be 0.0f.
   Return value:
   lhs.real()!=rhs.real() | | lhs.imag()!=rhs.imag()

9. istream _far & operator>>(istream _far & is, float_complex _far & x)
   Inputs **x** in a format of **u**, **(u)**, or **(u,v)** (**u**: real part, **v**: imaginary part).
   The input value is converted to **float_complex**.
   If **x** is input in a format other than the **u**, **(u)**, or **(u,v)** format, **is.setstate(ios_base::failbit)** is called.
   Return value:
   is

10. ostream _far & operator<<(ostream _far & os, const float_complex _far & x)
    Outputs **x** to **os**.
    The output format is **u**, **(u)**, or **(u,v)** (**u**: real part, **v**: imaginary part).
    Return value:
    os

11. float real(const float_complex _far & x)
    Acquires the real part.
    Return value:
    x.real()

12. float imag(const float_complex _far & x)
    Acquires the imaginary part.
    Return value:
        x.imag()

13. float abs(const float_complex _far & x)
    Calculates the absolute value.
    Return value:
        $(|x.real()|^2 + |x.imag()|^2)^{1/2}$

14. float arg(const float_complex _far & x)
    Calculates the phase angle.
    Return value:
        atan2f(x.imag(), x.real())

15. float norm(const float_complex _far & x)
    Calculates the absolute value of the square.
    Return value:
        $|x.real()|^2 + |x.imag()|^2$

16. float_complex conj(const float_complex _far & x)
    Calculates the conjugate complex number.
    Return value:
        float_complex(x.real(), (-1)*x.imag())

17. float_complex polar(const float _far & rho, const float _far & theta)
    Calculates the **float_complex** value for a complex number with size **rho** and phase angle (argument) **theta**.
    Return value:
        float_complex(rho*cosf(theta), rho*sinf(theta))

18. float_complex cos(const float_complex _far & x)
    Calculates the complex cosine.
    Return value:
        float_complex(cosf(x.real())*coshf(x.imag()), (-1)*sinf(x.real())*sinhf(x.imag()))

19. float_complex cosh(const float_complex _far & x)
    Calculates the complex hyperbolic cosine.
    Return value:
        cos(float_complex((-1)*x.imag(), x.real()))

20. float_complex exp(const float_complex _far & x)
    Calculates the exponent function.
    Return value:
        expf(x.real())*cosf(x.imag()),expf(x.real())*sinf(x.imag())

21. float_complex log(const float_complex _far & x)
    Calculates the natural logarithm (base e).
    Return value:
        float_complex(logf(abs(x)), arg(x))

22. float_complex log10(const float_complex _far & x)
Calculates the common logarithm (base 10).
Return value:
float_complex(log10f(abs(x)), arg(x)/logf(10))

23. float_complex pow(const float_complex _far & x, int y)
float_complex pow(const float_complex _far & x, const float _far & y)
float_complex pow(const float_complex _far & x, const float_complex _far & y)
float_complex pow(const float _far & x, const float_complex _far & y)
Calculates **x** to the **y**th power.
If **pow(0,0)**, a domain error will occur.
Return value:
If float_complex pow(const float_complex _far & x,const float_complex _far & y):
exp(y*logf(x))
Otherwise:
exp(y*log(x))

24. float_complex sin(const float_complex _far & x)
Calculates the complex sine.
Return value:
float_complex(sinf(x.real())*coshf(x.imag()), cosf(x.real())*sinhf(x.imag()))

25 float_complex sinh(const float_complex _far & x)
Calculates the complex hyperbolic sine.
Return value:float_complex(0,-1)*sin(float_complex((-1)*x.imag(),x.real()))

26. float_complex sqrt(const float_complex _far & x)
Calculates the square root within the right half space.
Return value:
float_complex(sqrtf(abs(x))*cosf(arg(x)/2), sqrtf(abs(x))*sinf(arg(x)/2))

27. float_complex tan(const float_complex _far & x)
Calculates the complex tangent.
Return value:
sin(x)/cos(x)

28. float_complex tanh(const float_complex _far & x)
Calculates the complex hyperbolic tangent.
Return value:
sinh(x)/cosh(x)

(c) double_complex Class

Type	Definition Name	Description
Type	value_type	**double** type
Variable	_re	Defines the real part of **double** precision
	_im	Defines the imaginary part of **double** precision
Function	double_complex(double re = 0.0, double im = 0.0)	Constructor
	double_complex(const float_complex _far &)	
	double real() const	Acquires the real part
	double imag() const	Acquires the imaginary part
	double_complex _far & operator=(double rhs)	Copies **rhs** to the real part 0.0 is assigned to the imaginary part
	double_complex _far & operator+=(double rhs)	Adds **rhs** to the real part of ***this** and stores the sum in ***this**
	double_complex _far & operator-=(double rhs)	Subtracts **rhs** from the real part of ***this** and stores the difference in ***this**.
	double_complex _far & operator*=(double rhs)	Multiplies ***this** by **rhs** and stores the product in ***this**
	double_complex _far & operator/=(double rhs)	Divides ***this** by **rhs** and stores the quotient in ***this**
	double_complex _far & operator=(const double_complex _far & rhs)	Copies **rhs**
	double_complex _far & operator+=(const double_complex _far & rhs)	Adds **rhs** to ***this** and stores the sum in ***this**
	double_complex _far & operator-=(const double_complex _far & rhs)	Subtracts **rhs** from ***this** and stores the difference in ***this**
	double_complex _far & operator*=(const double_complex _far & rhs)	Multiplies ***this** by **rhs** and stores the product in ***this**
	double_complex _far & operator/=(const double_complex _far & rhs)	Divides ***this** by **rhs** and stores the quotient in ***this**

1. double_complex::double_complex(double re = 0.0, double im = 0.0)
   Constructor of class **double_complex**.
   The initial settings are as follows:
   _re = re;
   _im = im;

2. double_complex::double_complex(const float_complex _far &)
   Constructor of class **double_complex**.
   The initial settings are as follows:
   _re = (double)rhs.real();
   _im = (double)rhs.imag();

3. double double_complex::real() const
   Acquires the real part.
   Return value:
   this->_re

4. double double_complex::imag() const
   Acquires the imaginary part.
   Return value:
   this->_im

5. double_complex _far & double_complex::operator=(double rhs)
   Copies **rhs** to the real part (**_re**).
   0.0 is assigned to the imaginary part (**_im**).
   Return value:
   ***this**

6. double_complex _far & double_complex::operator+=(double rhs)
   Adds **rhs** to the real part (**_re**) and stores the result in the real part (**_re**).
   The value of the imaginary part (**_im**) does not change.
   Return value:
   ***this**

7. double_complex _far & double_complex::operator-=(double rhs)
   Subtracts **rhs** from the real part (**_re**) and stores the result in the real part (**_re**).
   The value of the imaginary part (**_im**) does not change.
   Return value:
   ***this**

8. double_complex _far & double_complex::operator*=(double rhs)
   Multiplies ***this** by **rhs** and stores the result in ***this**.
   (_re=_re*rhs, _im=_im*rhs)
   Return value:
   ***this**

9. double_complex _far & double_complex::operator/=(double rhs)
   Divides ***this** by **rhs** and stores the result in ***this**.
   (_re=_re/rhs, _im=_im/rhs)
   Return value:
   ***this**

10. double_complex _far & double_complex::operator=(const double_complex _far & rhs)
    Copies **rhs** to ***this**.
    Return value:
    ***this**

11. double_complex _far & double_complex::operator+=(const double_complex _far & rhs)
    Adds **rhs** to ***this** and stores the result in ***this**.
    Return value:
    ***this**

12. double_complex _far & double_complex::operator-=(const double_complex _far & rhs)
    Subtracts **rhs** from ***this** and stores the result in ***this**.
    Return value:
        ***this**

13. double_complex _far & double_complex::operator*=(const double_complex _far & rhs)
    Multiplies ***this** by **rhs** and stores the result in ***this**.
    Return value:
        ***this**

14. double_complex _far & double_complex::operator/=(const double_complex _far & rhs)
    Divides ***this** by **rhs** and stores the result in ***this**.
    Return value:
        ***this**

(d) double_complex Non-Member Function

Type	Definition Name	Description
Function	double_complex operator+(const double_complex _far & lhs)	Performs unary + operation of **lhs**
	double_complex operator+(const double_complex _far & lhs, const double_complex _far & rhs)	Adds **rhs** to **lhs** and stores the sum in **lhs**
	double_complex operator+(const double_complex _far & lhs, const double _far & rhs)	
	double_complex operator+(const double _far & lhs, const double_complex _far & rhs)	
	double_complex operator-(const double_complex _far & lhs)	Performs unary - operation of **lhs**
	double_complex operator-(const double_complex _far & lhs, const double_complex _far & rhs)	Subtracts **rhs** from **lhs** and stores the difference in **lhs**
	double_complex operator-(const double_complex _far & lhs, const double _far & rhs)	
	double_complex operator-(const double _far & lhs, const double_complex _far & rhs)	
	double_complex operator*(const double_complex _far & lhs, const double_complex _far & rhs)	Multiples **lhs** by **rhs** and stores the product in **lhs**
	double_complex operator*(const double_complex _far & lhs, const double _far & rhs)	
	double_complex operator*(const double _far & lhs, const double_complex _far & rhs)	
	double_complex operator/(const double_complex _far & lhs, const double_complex _far & rhs)	Divides **lhs** by **rhs** and stores the quotient in **lhs**
	double_complex operator/(const double_complex _far & lhs, const double _far & rhs)	
	double_complex operator/(const double _far & lhs, const double_complex _far & rhs)	

Type	Definition Name	Description
Function	bool operator==(const double_complex _far & lhs, const double_complex _far & rhs)	Compares the real part of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**
	bool operator==(const double_complex _far & lhs, const double _far & rhs)	
	bool operator==(const double _far & lhs, const double_complex _far & rhs)	
	bool operator!=(const double_complex _far & lhs, const double_complex _far & rhs)	Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**
	bool operator!=(const double_complex _far & lhs, const double _far & rhs)	
	bool operator!=(const double _far & lhs, const double_complex _far & rhs)	
	istream _far & operator>>(istream _far & is, double_complex _far & x)	Inputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part)
	ostream _far & operator<<(ostream _far & os, const double_complex _far & x)	Outputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part)
	double real(const double_complex _far & x)	Acquires the real part
	double imag(const double_complex _far & x)	Acquires the imaginary part
	double abs(const double _complex _far & x)	Calculates the absolute value
	double arg(const double_complex _far & x)	Calculates the phase angle
	double norm(const double_complex _far & x)	Calculates the absolute value of the square
	double_complex conj(const double_complex _far & x)	Calculates the conjugate complex numbe
	double_complex polar(const double _far & rho, const double _far & theta)	Calculates the **double_complex** value for a complex number with size **rho** and phase angle **theta**
	double_complex cos(const double_complex _far & x)	Calculates the complex cosine
	double_complex cosh(const double_complex _far & x)	Calculates the complex hyperbolic cosine
	double_complex exp(const double_complex _far &)	Calculates the exponent function

Type	Definition Name	Description
Function	double_complex log(const double_complex _far & x)	Calculates the natural logarithm
	double_complex log10(const double_complex _far & x)	Calculates the common logarithm
	double_complex pow(const double_complex _far & x, int y)	Calculates **x** to the **y**th power
	double_complex pow(const double_complex _far & x, const double _far & y)	
	double_complex pow(const double_complex _far & x, const double_complex _far & y)	
	double_complex pow(const double _far & x, const double_complex _far & y)	
	double_complex sin(const double_complex _far & x)	Calculates the complex sine
	double_complex sinh(const double_complex _far & x)	Calculates the complex hyperbolic sine
	double_complex sqrt(const double_complex _far & x)	Calculates the square root within the right half space
	double_complex tan(const double_complex _far & x)	Calculates the complex tangent
	double_complex tanh(const double_complex _far & x)	Calculates the complex hyperbolic tangent

1. double_complex operator+(const double_complex _far & lhs)
   Performs unary + operation of **lhs**.
   Return value:
   lhs

2. double_complex operator+(const double_complex _far & lhs, const double_complex _far & rhs)
   double_complex operator+(const double_complex _far & lhs, const double _far & rhs)
   double_complex operator+(const double _far & lhs, const double_complex _far & rhs)
   Adds **lhs** to **rhs** and stores the result in **lhs**.
   Return value:
   double_complex(lhs)+=rhs

3. double_complex operator-(const double_complex _far & lhs)
   Performs unary - operation of **lhs**.
   Return value:
   double_complex(-lhs.real(), -lhs.imag())

4. double_complex operator-(const double_complex _far & lhs, const double_complex _far & rhs)
double_complex operator-(const double_complex _far & lhs, const double _far & rhs)
double_complex operator-(const double _far & lhs, const double_complex _far & rhs)
Subtracts **rhs** from **lhs** and stores the result in **lhs**.
Return value:
double_complex(lhs)-=rhs

5. double_complex operator*(const double_complex _far & lhs, const double_complex _far & rhs)
double_complex operator*(const double_complex _far & lhs, const double _far & rhs)
double_complex operator*(const double _far & lhs, const double_complex _far & rhs)
Multiplies **lhs** by **rhs** and stores the result in **lhs**.
Return value:
double_complex(lhs)*=rhs

6. double_complex operator/(const double_complex _far & lhs, const double_complex _far & rhs)
double_complex operator/(const double_complex _far & lhs, const double _far & rhs)
double_complex operator/(const double _far & lhs, const double_complex _far & rhs)
Divides **lhs** by **rhs** and stores the result in **lhs**.
Return value:
double_complex(lhs)/=rhs

7. bool operator==(const double_complex _far & lhs, const double_complex _far & rhs)
bool operator==(const double_complex _far & lhs, const double _far & rhs)
bool operator==(const double _far & lhs, const double_complex _far & rhs)
Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **double** type parameter, the imaginary part is assumed to be 0.0.
Return value:
hs.real()==rhs.real() && lhs.imag()==rhs.imag()

8. bool operator!=(const double_complex _far & lhs, const double_complex _far & rhs)
bool operator!=(const double_complex _far & lhs, const double _far & rhs)
bool operator!=(const double _far & lhs, const double_complex _far & rhs)
Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **double** type parameter, the imaginary part is assumed to be 0.0.
Return value:
lhs.real()!=rhs.real() | | lhs.imag()!=rhs.imag()

9. istream _far & operator>>(istream _far & is, double_complex _far & x)
Inputs complex number **x** in a format of **u**, **(u)**, or **(u,v)** (**u**: real part, **v**: imaginary part).
The input value is converted to **double_complex**.
If **x** is input in a format other than the **u**, **(u)**, or **(u,v)** format, **is.setstate(ios_base::failbit)** is called.
Return value:
is

10. ostream _far & operator<<(ostream _far & os, const double_complex _far & x)
Outputs **x** to **os**.
The output format is **u**, **(u)**, or **(u,v)** (**u**: real part, **v**: imaginary part).
Return value:
os

11. double real(const double_complex _far & x)
    Acquires the real part.
    Return value:
        x.real()

12. double imag(const double_complex _far & x)
    Acquires the imaginary part.
    Return value:
        x.imag()

13. double abs(const double_complex _far & x)
    Calculates the absolute value.
    Return value:
        $(|x.real()|^2 + |x.imag()|^2)^{1/2}$

14. double arg(const double_complex _far & x)
    Calculates the phase angle.
    Return value:
        atan2(x.imag(), x.real())

15. double norm(const double_complex _far & x)
    Calculates the absolute value of the square.
    Return value:
        $|x.real()|^2 + |x.imag()|^2$

16. double_complex conj(const double_complex _far & x)
    Calculates the conjugate complex number.
    Return value:
        double_complex(x.real(), (-1)*x.imag())

17. double_complex polar(const double _far & rho, const double _far & theta)
    Calculates the **double_complex** value for a complex number with size **rho** and phase angle (argument) **theta**.
    Return value:
        double_complex(rho*cos(theta), rho*sin(theta))

18. double_complex cos(const double_complex _far & x)
    Calculates the complex cosine.
    Return value:
        double_complex(cos(x.real())*cosh(x.imag()), (-1)*sin(x.real())*sinh(x.imag()))

19. double_complex cosh(const double_complex _far & x)
    Calculates the complex hyperbolic cosine.
    Return value:
        cos(double_complex((-1)*x.imag(), x.real()))

20. double_complex exp(const double_complex _far & x)
    Calculates the exponent function.
    Return value:
        exp(x.real())*cos(x.imag()),exp(x.real())*sin(x.imag())

21. double_complex log(const double_complex _far & x)
    Calculates the natural logarithm (base e).
    Return value:
    double_complex(log(abs(x)), arg(x))

22. double_complex log10(const double_complex _far & x)
    Calculates the common logarithm (base 10).
    Return value:
    double_complex(log10(abs(x)), arg(x)/log(10))

23. double_complex pow(const double_complex _far & x, int y)
    double_complex pow(const double_complex _far & x, const double _far & y)
    double_complex pow(const double_complex _far & x, const double_complex _far & y)
    double_complex pow(const double _far & x, const double_complex _far & y)
    Calculates x to the yth power.
    If pow(0,0), a domain error will occur.
    Return value:
    exp(y*log(x))

24. double_complex sin(const double_complex _far & x)
    Calculates the complex sine
    Return value:
    double_complex(sin(x.real())*cosh(x.imag()), cos(x.real())*sinh(x.imag()))

25. double_complex sinh(const double_complex _far & x)
    Calculates the complex hyperbolic sine
    Return value:
    double_complex(0,-1)*sin(double_complex((-1)*x.imag(),x.real()))

26. double_complex sqrt(const double_complex _far & x)
    Calculates the square root within the right half space
    Return value:
    double_complex(sqrt(abs(x))*cos(arg(x)/2), sqrt(abs(x))*sin(arg(x)/2))

27. double_complex tan(const double_complex _far & x)
    Calculates the complex tangent.
    Return value:
    sin(x)/cos(x)

28. double_complex tanh(const double_complex _far & x)
    Calculates the complex hyperbolic tangent.
    Return value:
    sinh(x)/cosh(x)

(5) String Handling Class Library

The header file for the string handling class library is as follows:

- <string>
  Defines class **string**.

This class has no derivation.

(a) string Class

Type	Definition Name	Description
Type	iterator	**char _far** *type
	const_iterator	**const char _far *** type
Constant	npos	Maximum string length (**UNIT_MAX** characters)
Variable	s_ptr	Pointer to the memory area where the string is stored by the object
	s_len	The length of the string stored by the object
	s_res	Size of the allocated memory area to store string by the object
Function	string(void)	Constructor
	string::string(const string _far & str, size_t pos = 0, size_t n = npos)	
	string::string(const char _far * str, size_t n)	
	string::string(const char _far * str)	
	string::string(size_t n, char c)	
	~string()	Destructor
	string _far & operator=(const string _far & str)	Assigns **str**
	string _far & operator=(const char _far * str)	
	string _far & operator=(char c)	Assigns **c**
	iterator begin()	Calculates the start pointer of the string
	const_iterator begin() const	
	iterator end()	Calculates the end pointer of the string
	const_iterator end() const	
	size_t size() const	Calculates the length of the stored string
	size_t length() const	
	size_t max_size() const	Calculates the size of the allocated memory area
	void resize(size_t n, char c)	Changes the storable string length to **n**
	void resize(size_t n)	Changes the storable string length to **n**

Type	Definition Name	Description
Function	size_t capacity() const	Calculates the size of the allocated memory area
	void reserve(size_t res_arg = 0)	Performs re-allocation of the memory area
	void clear()	Clears the stored string
	bool empty() const	Checks whether the stored string length is 0
	const char _far & operator[](size_t pos) const	References **s_ptr[pos]**
	char _far & operator[](size_t pos)	
	const char _far & at(size_t pos) const	
	char _far & at(size_t pos)	
	string _far & operator+=(const string _far & str)	Adds string **str**
	string _far & operator+=(const string _far & str)	
	string _far & operator+=(char c)	Adds character **c**
	string _far & append(const string _far & str)	Adds string **str**
	string _far & append(const char _far * str)	
	string _far & append(const string _far & str, size_t pos, size_t n)	Adds **n** characters of string **str** at object position **pos**
	string _far & append(const char _far * str, size_t n)	Adds **n** characters to string **str**
	string _far & append(size_t n, char c)	Adds **n** characters, each of which is **c**
	string _far & assign(const string _far & str)	Assigns string **str**
	string _far & assign(const char _far * str)	
	string _far & assign(const string _far & str, size_t pos, size_t n)	Add **n** characters to string **str** at position **pos**
	string _far & assign(const char _far * str, size_t n)	Assigns **n** characters of string **str**
	string _far & assign(size_t n, char c)	Assigns **n** characters, each of which is **c**
	string _far & insert(size_t pos1, const string _far & str)	Inserts string **str** to position **pos1**
	string _far & insert(size_t pos1, const string _far & str, size_t pos2, size_t n)	Inserts **n** characters starting from position **pos2** of string **str** to position **pos1**
	string _far & insert(size_t pos, const char _far * str, size_t n)	Inserts **n** characters of string **str** to position **pos**
	string _far & insert(size_t pos, const char _far * str)	Inserts string **str** to position **pos**

Type	Definition Name	Description
Function	string _far & insert(size_t pos, size_t n, char c)	Inserts a string of **n** characters, each of which is **c**, to position **pos**
	iterator insert(iterator p, char c = char())	Inserts character **c** before the string specified by **p**
	void insert(iterator p, size_t n, char c)	Inserts **n** characters, each of which is **c**, before the character specified by **p**
	string _far & erase(size_t pos = 0, size_t n = npos)	Deletes **n** characters from position **pos**
	iterator erase(iterator position)	Deletes the character referenced by **position**
	iterator erase(iterator first, iterator last)	Deletes the characters in range [**first**, **last**]
	string _far & replace(size_t pos1, size_t n1, const string _far & str)	Replaces the string of **n1** characters starting from position **pos1** with string **str**
	string _far & replace(size_t pos1, size_t n1, const char _far * str)	
	string _far & replace(size_t pos1, size_t n1, const string _far & str, size_t pos2, size_t n2)	Replaces the string of **n1** characters starting from position **pos1** with string of **n2** characters from position **pos2** of **str**
	string _far & replace(size_t pos, size_t n1, const char _far * str, size_t n2)	Replaces the string of **n1** characters starting from position **pos** with string **str** of **n2** characters
	string _far & replace(size_t pos, size_t n1, size_t n2, char c)	Replaces the string of **n1** characters starting from position **pos** with **n2** characters, each of which is **c**
	string _far & replace(iterator i1, iterator i2, const string _far & str)	Replaces the string from position **i1** to **i2** with string **str**
	string _far & replace(iterator i1, iterator i2, const char _far * str)	

Type	Definition Name	Description
Function	string _far & replace(iterator i1, iterator i2, const char _far * str,, size_t n)	Replaces the string from position **i1** to **i2** with **n** characters of string **str**
	string _far & replace(iterator i1, iterator i2, size_t n, char c)	Replaces the string from position **i1** to **i2** with **n** characters, each of which is **c**
	size_t copy(char _far * str,, size_t n, size_t pos = 0) const	Copies the first **n** characters of string **str** to position **pos**
	void swap(string _far & str)	Swaps ***this** with string **str**
	const char _far * c_str() const	References the pointer to the memory area where the string is stored
	const char _far * data() const	
	size_t find(const string _far & str, size_t pos = 0) const	Finds the position where the string same as string **str** first appears after position **pos**
	size_t find(const char _far * str,, size_t pos = 0) const	
	size_t find(const char _far * str,, size_t pos, size_t n) const	Finds the position where the string same as **n** characters of **str** first appears after position **pos**
	size_t find(char c, size_t pos = 0) const	Finds the position where character **c** first appears after position **pos**
	size_t rfind(const string _far & str, size_t pos = npos) const	Finds the position where a string same as string **str** appears most recently before position **pos**
	size_t rfind(const char _far * str,, size_t pos = npos) const	
	size_t rfind(const char _far * str,, size_t pos, size_t n) const	Finds the position where the string same as **n** characters of **str** appears most recently before position **pos**
	size_t rfind(char c, size_t pos = npos) const	Finds the position where character **c** appears most recently before position **pos**

Type	Definition Name	Description
Function	size_t find_first_of(const string _far & str, size_t pos = 0) const	Finds the position where any character included in string **str** first appears after position **pos**
	size_t find_first_of(const char _far * str , size_t pos = 0) const	
	size_t find_first_of(const char _far * str , size_t pos, size_t n) const	Finds the position where any character included in **n** characters of string **str** first appears after position **pos**
	size_t find_first_of(char c, size_t pos = 0) const	Finds the position where character **c** first appears after position **pos**
	size_t find_last_of(const string _far & str, size_t pos = npos) const	Finds the position where any character included in string **str** appears most recently before position **pos**
	size_t find_last_of(const char _far * str , size_t pos = npos) const	
	size_t find_last_of(const char _far * str , size_t pos, size_t n) const	Finds the position where any character included in **n** characters of string **str** appears most recently before position **pos**
	size_t find_last_of(char c, size_t pos = npos) const	Finds the position where character **c** appears most recently before position **pos**
	size_t find_first_not_of(const string _far & str, size_t pos = 0) const	Finds the position where a character different from any character included in string **str** first appears after position **pos**
	size_t find_first_not_of(const char _far * str , size_t pos = 0) const	
	size_t find_first_not_of(const char _far * str , size_t pos, size_t n)	Finds the position where a character different from any character in the first **n** characters of string **str** appears after position **pos**.
	size_t find_first_not_of(char c, size_t pos = 0) const	Finds the position where a character different from **c** first appears after position **pos**
	size_t find_last_not_of(const string _far & str, size_t pos = npos) const	Finds the position where a character different from any character included in string **str** appears most recently before position **pos**
	size_t find_last_not_of(const char _far * str , size_t pos = npos) const	

Type	Definition Name	Description
Function	size_t find_last_not_of(const char _far * str, size_t pos, size_t n) const	Finds the position where a character different from any character in the first **n** characters of string **str** appears most recently before position **pos**.
	size_t find_last_not_of(char c, size_t pos = npos) const	Finds the position where a character different from **c** appears most recently before position **pos**
	string substr(size_t pos = 0, size_t n = npos) const	Creates an object from a string in the range [**pos**,**n**] of the stored string
	int compare(const string _far & str) const	Compares the string with string **str**
	int compare(size_t pos1, size_t n1, const string _far & str) const	Compares **n1** characters from position **pos1** of ***this** with **str**
	int compare(size_t pos1, size_t n1, const string _far & str, size_t pos2, size_t n2) const	Compares the string of **n1** characters from position **pos1** with the string of **n2** characters from position **pos2** of string **str**
	int compare(const char _far * str) const	Compares ***this** with string **str**
	int compare(size_t pos1, size_t n1, const char _far * str, size_t n2 = npos) const	Compares the string of **n1** characters from position **pos1** with **n2** characters of string **str**

1. string::string(void)
   Sets as follows:
   s_ptr = 0;
   s_len = 0;
   s_res = 1;

2. string::string(const string _far & str, size_t pos = 0, size_t n = npos)
   Copies **str**. Note that **s_len** will be the smaller value of **n** and **s_len**.

3. string::string(const char _far * str, size_t n)
   Sets as follows:
   s_ptr = **str**;
   s_len = **n**;
   s_res = **n** + 1;

4. string::string(const char _far * str)
   Sets as follows:
   s_ptr = **str**;
   s_len = length of string **str**;
   s_res = length of string **str** + 1;

5. string::string(size_t n, char c)
   Sets as follows:
   s_ptr = string of **n** characters, each of which is **c**
   s_len = **n**;
   s_res = **n** + 1;

6. string::~string()
   Destructor of class **string**.
   Deallocates the memory area where the string is stored.

7. string _far & string::operator=(const string _far & str)
   Assigns the data of **str**.
   Return value:
   ***this**

8. string _far & string::operator=(const char _far * str)
   Creates a **string** object from **str** and assigns its data to the **string** object.
   Return value:
   ***this**

9. string _far & string::operator=(char c)
   Creates a **string** object from **c** and assigns its data to the **string** object.
   Return value:
   ***this**

10. string::iterator string::begin()
    string::const_iterator string::begin() const
    Calculates the start pointer of the string.
    Return value:
    Start pointer of the string

11. string::iterator string::end()
    string::const_iterator string::end() const
    Calculates the end pointer of the string.
    Return value:
    End pointer of the string

12. size_t string::size() const
    size_t string::length() const
    Calculates the length of the stored string.
    Return value:
    Length of the stored string

13. size_t string::max_size() const
    Calculates the size of the allocated memory area.
    Return value:
        Size of the allocated area

14. void string::resize(size_t n, char c)
    Changes the number of characters in the string that can be stored by the object to **n**.
    If **n<=size()**, replaces the string with the original string with length **n**.
    If **n>size()**, replaces the string with a string that has **c** appended to the end so that the length will be equal to **n**.
    The length must be **n<=max_size()**.
    If n>**max_size()**, the string length is **n=max_size()**.

15. void string::resize(size_t n)
    Changes the number of characters in the string that can be stored by the object to **n**.
    If **n<=size()**, replaces the string with the original string with length **n**.
    The length must be **n<=max_size**.

16. size_t string::capacity() const
    Calculates the size of the allocated memory area.
    Return value:
        Size of the allocated memory area

17. void string::reserve(size_t res_arg = 0)
    Re-allocates the memory area.
    After **reserve()**, **capacity()** will be equal to or larger than the **reserve()** parameter.
    When the memory area is re-allocated, all references, pointers, and **iterator** that references the elements of the numeric sequence become invalid.

18. void string::clear()
    Clears the stored string.

19. bool string::empty() const
    Checks whether the number of characters in the stored string is 0.
    Return value:
        If the length of the stored string is 0:
            true
        If the length of the stored string is not zero:
            false

20. const char _far & string::operator[](size_t pos) const
    char _far & string::operator[](size_t pos)
    const char _far & string::at(size_t pos) const
    char _far & string::at(size_t pos)
    References **s_ptr[pos]**.
    Return value:
        If **n< s_len**:
            s_ptr [pos]
        If **n>= s_len**:
            '\0'

21. string _far & string::operator+=(const string _far & str)
    Appends the string stored in **str** to the object.
    Return value:
        ***this**

22. string _far & string::operator+=(const char _far * str)
    Creates a **string** object from **str** and adds the string to the object.
    Return value:
        ***this**

23. string _far & string::operator+=(char c)
    Creates a **string** object from **c** and adds the string to the object.
    Return value:
        ***this**

24. string _far & string::append(const string _far & str)
    string _far & string::append(const char _far * str)
    Appends string **str** to the object.
    Return value:
        ***this**

25. string _far & string::append(const string _far & str, size_t pos, size_t n)
    Appends **n** characters of string **str** to the object position **pos**.
    Return value:
        ***this**

26. string _far & string::append(const char _far * str, size_t n)
    Appends **n** characters of string **str** to the object.
    Return value:
        ***this**

27. string _far & string::append(size_t n, char c)
    Appends **n** characters, each of which is **c**, to the object.
    Return value:
        ***this**

28. string _far & string::assign(const string _far & str)
    string _far & string::assign(const char _far * str)
    Assigns string **str**.
    Return value:
        ***this**

29. string _far & string::assign(const string _far & str, size_t pos, size_t n)
    Assigns **n** characters of string **str** to position **pos**.
    Return value:
        ***this**

30. string _far & string::assign(const char _far * str, size_t n)
    Assigns **n** characters of string **str**.
    Return value:
        ***this**

31. string _far & string::assign(size_t n, char c)
    Assigns **n** characters, each of which is **c**.
    Return value:
        ***this**

32. string _far & string::insert(size_t pos1, const string _far & str)
    Inserts string **str** to position **pos1**.
    Return value:
        ***this**

33. string _far & string::insert(size_t pos1, const string _far & str, size_t pos2, size_t n)
    Inserts **n** characters starting from position **pos2** of string **str** to position **pos1**.
    Return value:
        ***this**

34. string _far & string::insert(size_t pos, const char _far * str, size_t n)
    Inserts **n** characters of string **str** to position **pos**.
    Return value:
        ***this**

35. string _far & string::insert(size_t pos, const char _far * str)
    Inserts string **str** to position **pos**.
    Return value:
        ***this**

36. string _far & string::insert(size_t pos, size_t n, char c)
    Inserts a string of **n** characters, each of which is **c**, to position **pos**.
    Return value:
        ***this**

37. string::iterator string::insert(iterator p, char c = char())
    Inserts character **c** before the string specified by **p**.
    Return value:
        The inserted character

38. void string::insert(iterator p, size_t n, char c)
    Inserts **n** characters, each of which is **c**, before the character specified by **p**.

39. string _far & string::erase(size_t pos = 0, size_t n = npos)
    Deletes **n** characters starting from position **pos**.
    Return value:
        ***this**

40. iterator string::erase(iterator position)
    Deletes the character referenced by **position**.
    Return value:
        If the next **iterator** of the element to be deleted exists:
            The next **iterator** of the deleted element
        If the next **iterator** of the element to be deleted does not exist:
            end()

41. iterator string::erase(iterator first, iterator last)
Deletes the characters in range [**first**, **last**].
Return value:
If the next **iterator** of **last** exists:
The next **iterator** of **last**
If the next **iterator** of **last** does not exist:
end()

42. string _far & string::replace(size_t pos1, size_t n1, const string _far & str)
string _far & string::replace(size_t pos1, size_t n1, const char _far * str)
Replaces the string of **n1** characters starting from position **pos1** with string **str**.
Return value:
***this**

43. string _far & string::replace(size_t pos1, size_t n1, const string _far & str, size_t pos2, size_t n2)
Replaces the string of **n1** characters starting from position **pos1** with the string of **n2** characters starting from position **pos2** in string **str**.
Return value:
***this**

44. string _far & string::replace(size_t pos, size_t n1, const char _far * str, size_t n2)
Replaces the string of **n1** characters starting from position **pos1** with **n2** characters of string **str**.
Return value:
***this**

45. string _far & string::replace(size_t pos, size_t n1, size_t n2, char c)
Replaces the string of **n1** characters starting from position **pos** with **n2** characters, each of which is **c**.
Return value:
***this**

46. string _far & string::replace(iterator i1, iterator i2, const string _far & str)
string _far & string::replace(iterator i1, iterator i2, const char _far * str)
Replaces the string from position **i1** to **i2** with string **str**.
Return value:
***this**

47. string _far & string::replace(iterator i1, iterator i2, const char _far * str, size_t n)
Replaces the string from position **i1** to **i2** with **n** characters of string **str**
Return value:
***this**

48. string _far & string::replace(iterator i1, iterator i2, size_t n, char c)
Replaces the string from position **i1** to **i2** with **n** characters, each of which is **c**.
Return value:
***this**

49. size_t string::copy(char* _far str, size_t n, size_t pos = 0) const
Copies **n** characters of string **str** to position **pos**.
Return value:
rlen

50. void string::swap(string _far & str)
Swaps ***this** with string **str**.

51. const char _far * string::c_str() const
const char _far * string::data() const
References the pointer to the memory area where the string is stored.
Return value:
s_ptr

52. size_t string::find(const string _far & str, size_t pos = 0) const
size_t string::find(const char _far * str, size_t pos = 0) const
Finds the position where the string same as string **str** first appears after position **pos**.
Return value:
Offset of string

53. size_t string::find(const char _far * str, size_t pos, size_t n) const
Finds the position where the string same as **n** characters of string **str** first appears after position **pos**.
Return value:
Offset of string

54. size_t string::find(char c, size_t pos = 0) const
Finds the position where character **c** first appears after position **pos**.
Return value:
Offset of string

55. size_t string::rfind(const string _far & str, size_t pos = npos) const
size_t string::rfind(const char _far * str, size_t pos = npos) const
Finds the position where a string same as string **str** appears most recently before position **pos**.
Return value:
Offset of string

56. size_t string::rfind(const char _far * str, size_t pos, size_t n) const
Finds the position where the string same as **n** characters of string **str** appears most recently before position **pos**.
Return value:
Offset of string

57. size_t string::rfind(char c, size_t pos = npos) const
Finds the position where character **c** appears most recently before position **pos**.
Return value:
Offset of string

58. size_t string::find_first_of(const string _far & str, size_t pos = 0) const
size_t string::find_first_of(const char _far * str, size_t pos = 0) const
Finds the position where any character included in string **str** first appears after position **pos**.
Return value:
Offset of string

59. size_t string::find_first_of(const char _far * str, size_t pos, size_t n) const
    Finds the position where any character included in **n** characters of string **str** first appears after position **pos**.
    Return value:
        Offset of string

60. size_t string::find_first_of(char c, size_t pos = 0) const
    Finds the position where character **c** first appears after position **pos**.
    Return value:
        Offset of string

61. size_t string::find_last_of(const string _far & str, size_t pos = npos) const
    size_t string::find_last_of(const char _far * str, size_t pos = npos) const
    Finds the position where any character included in string **str** appears most recently before position **pos**.
    Return value:
        Offset of string

62. size_t string::find_last_of(const char _far * str, size_t pos, size_t n) const
    Finds the position where any character included in **n** characters of string **str** appears most recently before position **pos**.
    Return value:
        Offset of string

63. size_t string::find_last_of(char c, size_t pos = npos) const
    Finds the position where character **c** appears most recently before position **pos**.
    Return value:
        Offset of string

64. size_t string::find_first_not_of(const string _far & str, size_t pos = 0) const
    size_t string::find_first_not_of(const char _far * str, size_t pos = 0) const
    Finds the position where a character different from any character included in string **str** first appears after position **pos**.
    Return value:
        Offset of string

65. size_t string::find_first_not_of(const char _far * str, size_t pos, size_t n) const
    Finds the position where a character different from any character in the first **n** characters of string **str** first appears after position **pos**.
    Return value:
        Offset of string

66. size_t string::find_first_not_of(char c, size_t pos = 0) const
    Finds the position where a character different from character **c** first appears after position **pos**.
    Return value:
        Offset of string

67. size_t string::find_last_not_of(const string _far & str, size_t pos = npos) const
size_t string::find_last_not_of(const char _far * str, size_t pos = npos) const
Finds the position where a character different from any character included in string **str** appears most recently before position **pos**.
Return value:
Offset of string

68. size_t string::find_last_not_of(const char _far * str, size_t pos, size_t n) const
Finds the position where a character different from any character in the first **n** characters of string **str** appears most recently before position **pos**.
Return value:
Offset of string

69. size_t string::find_last_not_of(char c, size_t pos = npos) const
Finds the position where a character different from character **c** appears most recently before position **pos**.
Return value:
Offset of string

70. string string::substr(size_t pos = 0, size_t n = npos) const
Creates an object from a string in the range [**pos**,**n**] of the stored string.
Return value:
Object with a string in the range [**pos**,**n**]

71. int string::compare(const string _far & str) const
Compares the string with string **str**.
Return value:
If the strings are the same:
0
If the strings are different:
1 when this->**s_len** > **str.s_len**,1 when this->**s_len** < **str.s_len**

72. int string::compare(size_t pos1, size_t n1, const string _far & str) const
Compares a string of **n1** characters starting from position **pos1** of ***this** with string **str**.
Return value:
If the strings are the same:
0
If the strings are different:
1 when **this->s_len** > **str.s_len**, -1 when **this->s_len** < **str.s_len**

73. int string::compare(size_t pos1, size_t n1, const string _far & str, size_t pos2, size_t n2) const
Compares a string of **n1** characters starting from position **pos1** with the string of **n2** characters from position **pos2** of string **str**.
Return value:
If the strings are the same:
0
If the strings are different:
1 when **this->s_len** > **str.s_len**, -1 when **this->s_len** < **str.s_len**

74. int string::compare(const char _far * str) const

Compares ***this** with string **str**.

Return value:

If the strings are the same:

0

If the strings are different:

1 when **this->s_len** > **str.s_len**, -1 when **this->s_len** < **str.s_len**

75. int string::compare(size_t pos1, size_t n1, const char _far * str, size_t n2 = npos) const

Compares the string of **n1** characters from position **pos1** with **n2** characters of string **str**.

Return value:

If the strings are the same:

0

If the strings are different:

1 when this->s_len > str.s_len, -1 when this->s_len < str.s_len

(b) string Class Manipulators

Type	Definition Name	Description
Function	string operator+(const string _far & lhs, const string _far & rhs)	Appends the string (or characters) of **rhs** to the string (or characters) of **lhs**, creates an object and stores the string in the object
	string operator+(const char _far * lhs, const string _far & rhs)	
	string operator+(char lhs, const string _far & rhs)	
	string operator+(const string _far & lhs, const char _far * rhs)	
	string operator+(const string _far & lhs, char rhs)	
	bool operator==(const string _far & lhs, const string _far & rhs)	Compares the string of **lhs** with the string of **rhs**
	bool operator==(const char _far * lhs, const string _far & rhs)	
	bool operator==(const string _far & lhs, const char _far * rhs)	
	bool operator!=(const string _far & lhs, const string _far & rhs)	Compares the string of **lhs** with the string of **rhs**
	bool operator!=(const char _far * lhs, const string _far & rhs)	
	bool operator!=(const string _far & lhs, const char _far * rhs)	
	bool operator<(const string _far & lhs, const string _far & rhs)	Compares the string length of **lhs** with the string length of **rhs**
	bool operator<(const char _far * lhs, const string _far & rhs)	
	bool operator<(const string _far & lhs, const char _far * rhs)	
	bool operator>(const string _far & lhs, const string _far & rhs)	Compares the string length of **lhs** with the string length of **rhs**
	bool operator>(const char _far * lhs, const string _far & rhs)	
	bool operator>(const string _far & lhs, const char _far * rhs)	
	bool operator<=(const string _far & lhs, const string _far & rhs)	Compares the string length of **lhs** with the string length of **rhs**
	bool operator<=(const char _far * lhs, const string _far & rhs)	
	bool operator<=(const string _far & lhs, const char _far * rhs)	
	bool operator>=(const string _far & lhs, const string _far & rhs)	Compares the string length of **lhs** with the string length of **rhs**
	bool operator>=(const char _far * lhs, const string _far & rhs)	
	bool operator>=(const string _far & lhs, const char _far * rhs)	

Type	Definition Name	Description
Function	void swap(string _far & lhs, string _far & rhs)	Swaps a string lhs and a string rhs.
	istream _far & operator>>(istream _far & is, string _far & str)	Retrieves a string into str.
	ostream _far & operator<<(ostream _far & os, const string _far & str)	Inserts a string.
	istream _far & getline(istream _far & is, string _far & str char delim)	Retrieves a string from ls and adds it to str. When a character 'delim' is detected in the middle, input is terminated.
	istream _far & getline(istream _far & is, string _far & str)	Retrieves a string from is and adds it to str. When a new-line character is detected in the middle, input is terminated.

1. string operator+(const string _far & lhs, const string _far & rhs)
   string operator+(const char _far * lhs, const string _far & rhs)
   string operator+(char lhs, const string _far & rhs)
   string operator+(const string _far & lhs, const char _far * rhs)
   string operator+(const string _far & lhs, char rhs)
   Appends the string (characters) of **lhs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.
   Return value:

   Object where the linked strings are stored

2. bool operator==(const string _far & lhs, const string _far & rhs)
   bool operator==(const char _far * lhs, const string _far & rhs)
   bool operator==(const string _far & lhs, const char _far * rhs)
   Compares the string of **lhs** with the string of **rhs**.
   Return value:

   If the strings are the same:

   true

   If the strings are different:

   false

3. bool operator!=(const string _far & lhs, const string _far & rhs)
   bool operator!=(const char _far * lhs, const string _far & rhs)
   bool operator!=(const string _far & lhs, const char _far * rhs)
   Compares the string of **lhs** with the string of **rhs**.
   Return value:

   If the strings are the same:

   false

   If the strings are different:

   true

4. bool operator<(const string _far & lhs, const string _far & rhs)
   bool operator<(const char _far * lhs, const string _far & rhs)
   bool operator<(const string _far & lhs, const char _far * rhs)
   Compares the string length of **lhs** with the string length of **rhs**.
   Return value:
   If lhs.s_len < rhs.s_len:
   true
   If lhs.s_len >= rhs.s_len:
   false

5. bool operator>(const string _far & lhs, const string _far & rhs)
   bool operator>(const char _far * lhs, const string _far & rhs)
   bool operator>(const string _far & lhs, const char _far * rhs)
   Compares the string length of **lhs** with the string length of **rhs**.
   Return value:
   If lhs.s_len > rhs.s_len:
   true
   If lhs.s_len <= rhs.s_len:
   false

6. bool operator<=(const string _far & lhs, const string _far & rhs)
   bool operator<=(const char _far * lhs, const string _far & rhs)
   bool operator<=(const string _far & lhs, const char _far * rhs)
   Compares the string length of **lhs** with the string length of **rhs**.
   Return value:
   If lhs.s_len <= rhs.s_len:
   true
   If lhs.s_len > rhs.s_len:
   false

7. bool operator>=(const string _far & lhs, const string _far & rhs)
   bool operator>=(const char _far * lhs, const string _far & rhs)
   bool operator>=(const string _far & lhs, const char _far * rhs)
   Compares the string length of **lhs** with the string length of **rhs**.
   Return value:
   If lhs.s_len >= rhs.s_len:
   true
   If lhs.s_len < rhs.s_len:
   false

8. void swap(string _far & lhs, string _far & rhs)
   Swaps the string of **lhs** with the string of **rhs**.

9. istream& operator>>(istream _far & is, string _far & str)
   Extracts a string to **str**.
   Return value:
   is

10. ostream& operator<<(ostream _far & os, const string _far & str)
    Inserts string **str**.
    Return value:
        os

11. istream& getline(istream _far & is, string _far & str, char delim)
    Extracts a string from **is** and appends it to **str**.
    If **delim** is found in the string, the input is stopped.
    Return value:
        is

12. istream& getline(istream _far & is, string _far & str)
    Extracts a string from **is** and appends it to **str**.
    If a new-line character is found, the input is stopped.
    Return value:
        is

Appendix F   Compiler Error Messages

F.1   Error Format and Error Levels

The error messages output in the form below and the contents of these errors are described here..
Error number (Error level) Error message
    Error details
    Solution
    Error details (part 2)
    Solution (part 2)
There are five different error levels, corresponding to different degrees of seriousness.

Error Level	Error Type	Description
(I)	Information	Processing is continued.
(W)	Warning	Processing is continued.
(E)	Error	Processing is interrupted.
(F)	Fatal	Processing is interrupted.
(-)	Internal	Processing is interrupted.

F.1.1   Command Input Format of the Compile Driver

`C1001 (W) Ignore option '-?'`

An unusable compile option -? is used.
Specify the correct compile option.

`C1002 (W) Ignore option 'option' is no effect when compiling C++`

An option that has no effect in C++ compilation is specified.
Delete the specified option when compiling C++.

`C1003 (W) Ignore option 'option' is no effect when compiling C`

An option that has no effect in C compilation is specified.
Delete the specified option when compiling C.

`C1004 (W) Nothing to compile, assemble or link`

No input files to compile, assemble or link are specified.
Specify the input files to compile, assemble or link on the command line.

`C1005 (W) Can't specified 'option A' with 'option B' option. 'option A' was ignored`

The specified option A is the one that cannot be specified simultaneously with the option B.
Be sure that the specified option A is not specified simultaneously with the option B.

`C1511 (W) #pragma pragma-name & HANDLER both specified`

Both #pragma pragma name and #pragma HANDLER are specified in one function.
Specify #pragma pragma name and #pragma HANDLER exclusive to each other.

`C1511 (W) #pragma pragma-name & INTERRUPT both specified`

Both #pragma pragma name and #pragma INTERRUPT are specified in one function.
Specify #pragma pragma name and #pragma INTERRUPT exclusive to each other.

`C1511 (W) #pragma pragma-name & TASK both specified`

Both #pragma pragma name and #pragma TASK are specified in one function.
Specify #pragma pragma name and #pragma TASK exclusive to each other.

`C1512 (W) #pragma pragma-name format error`

#pragma pragma name is erroneously written.
Follow the grammar of the manual as you write.

`C1512 (W) #pragma pragma-name format error, ignored`

#pragma pragma name is erroneously written. This line will be ignored.
Follow the grammar in the manual as you write.

`C1513 (W) #pragma JSRA illegal location, ignored`

#pragma JSRA is written in a function scope.
Write #pragma JSRA outside the function scope.

`C1513 (W) #pragma JSRW illegal location, ignored`

#pragma JSRW is written in a function scope.
Write #pragma JSRW outside the function scope.

`C1514 (W) #pragma pragma-name not function, ignored`

The name written in #pragma pragma name is not a function.
Write a function name for the subject to be operated on by #pragma.

`C1515 (W) #pragma pragma-name's function must be pre-declared, ignored`

The function specified by #pragma pragma name is not declared.
The function specified with #pragma pragma name must have its prototypes declared in advance.

`C1516 (W) #pragma pragma name's function must be prototyped, ignored`

The function specified with #pragma pragma name is not prototyped.
The function specified with #pragma pragma name must have its prototypes declared in advance.

`C1517 (W) #pragma pragma name's function return type invalid, ignored`

The function specified by #pragma pragma name includes an invalid type specified for its return value.
For the function's return value, specify the type other than struct, union, or double.

`C1518 (W) #pragma pragma-name variable initialized, initialization ignored`

The variable specified by #pragma pragma name is going to be initialized. Initialization will be ignored.
Delete either the #pragma pragma name or the initialization expression.

`C1527 (W) #pragma pragma-name variable must be far pointer for variable-name, ignored`

The variable declared in #pragma pragma name must be a far pointer. The #pragma declaration will be ignored.
To enable #pragma, declare the variable as a far pointer.

`C1528 (W) #pragma pragma-name variable must be unsigned int for variable-name, ignored`

The variable declared in #pragma pragma name must be unsigned int type. The #pragma declaration will be ignored.
To enable #pragma, declare the variable as unsigned int type.

`C1529 (W) #pragma pragma-name, register conflict`

In a #pragma pragma name declaration, the same register is used multiple times.
Be sure that one register is used only once.

`C1530 (W) #pragma pragma name, unknown register name used`

In a #pragma pragma name declaration, the string specifying a register is incorrect.
Follow the grammar of the manual as you write.

`C1531 (W) #pragma pragma-name variable must be pre-declared, ignored`

The variable declared in #pragma pragma name must have its type declared beforehand.
Declare the variable before #pragma.

`C1532 (W) #pragma ASM line too long, then cut`

The number of characters per line of 1,024 bytes writable in #pragma ASM is exceeded.
Write #pragma ASM in 1,024 bytes or less.

`C1533 (W) #pragma directive conflict`

Multiple #pragma directives that cannot be specified for one function at the same time are specified.
Delete the #pragma directives that are not simultaneously specified from the declaration.

`C1534 (W) #pragma for non-function type can not use for function`

A #pragma, not the type specifiable for functions, is specified for a function.
Delete the #pragma.

`C1536 (W) #pragma PARAMETER function's address used`

The address of a function specified by #pragma PARAMETER is being referenced.
Do not reference function address.

`C1537 (W) #pragma SECTADDRESS's attribute format error, ignored`

The section attribute string in #pragma SECTADDRESS is incorrect.
Write the correct section attribute name.

`C1538 (W) #pragma pragma-name unknown switch, ignored`

An invalid switch is written in #pragma pragma name.
Specify the correct switch.

`C1538 (W) #pragma unknown switch, ignored`

An invalid switch is specified for #pragma. The #pragma declaration will be ignored.
Specify the correct switch.

`C1539 (W) #pragma 'pragma-name' is aleady setted to 'value'`

The #pragma has the 'value' already set for it with the same 'pragma name.'
Do not set a different value for one variable or function a number of times in the same pragma.

`C1541 (W) invalid #pragma pragma-name`

The #pragma EQU is written erroneously. This line will be ignored.
Follow the grammar of the manual as you write.

`C1542 (W) invalid #pragma SECTION, unknown section base name`

The section name in #pragma SECTION is erroneous. The specifiable section names are data, bss, program, and rom. This line will be ignored.
Follow the grammar of the manual as you write.

`C1543 (W) Kanji in #pragma ADDRESS`

The #pragma ADDRESS written here includes kanji code. This line will be ignored.
Do not use kanji code in this declaration.

`C1543 (W) Kanji in #pragma BITADDRESS`

The #pragma BITADDRESS written here includes kanji code. This line will be ignored.
Do not use kanji code in this declaration.

`C1544 (W) this return type can not use for #pragma pragma-name, #pragma is ignored`

No 'pragma name' can be specified for the functions that return this type.
Do not specify #pragma or change the type of the function.

`C1545 (W) this variable's type is not match for 'register-name', #pragma 'pragma-name' is ignored`

The type of parameter and the register size do not match.
Make sure the type of parameter and the register size match.

`C1546 (W) unknown pragma pragma-specification used`

An unsupported #pragma is written.
Check the content of the #pragma. This warning is displayed only when the compile option -Wunknown_pragma (-WUP) or -Wall is specified.

`C1547 (W) OS version specifier conflict with another #pragma`

RTOS versions cannot coexist in #pragma.
Be sure the RTOS version is consistent.

`C1548 (W) cannot use SPECIAL PAGE number value, #pragma is ignored`

This value is out of the usable range of special page numbers.
Set a value usable for special pages.

`C1549 (W) function "function-name" in #pragma is not declared`

The function specified by #pragma is not declared.
Declare the function or delete #pragma.

`C1550 (W) #pragma DMAC variable must be unsigned long for variable, ignored`

The DMAC specified 'variable' must be unsigned long type.
Be sure the type of variable and the register name match.

`C1551 (W) #pragma DMAC variable must be far pointer to object for variable, ignored`

The DMAC specified 'variable' must be a far pointer that points to object type.
Be sure the type of variable and the register name match.

`C1571 (W) constant variable assignment`

An attempt is made to assign a value to the variable specified with const type qualifier.
Delete const from the variable declaration or stop the assignment.

`C1573 (W) octal constant is out of range`

The octal constant contains a character that cannot be used in octal representation.
Use numbers 0 to 7 to write octal constants.

`C1574 (W) integer constant is out of range`

The value of the integer constant exceeds the values representable by unsigned long long.
For the constant value, use a value representable by unsigned long long.

`C1575 (W) multi-character character constant`

A character constant containing more than one character is used.
If more than one character, use a wide character (L'xx').

`C1576 (W) hex character is out of range`

The hexadecimal escape sequence in a character constant is too long. Also, \ is followed by other than a hexadecimal character.
Cut the hexadecimal escape sequence shorter.

`C1577 (W) too big octal character`

The octal constant in a character constant or string exceeds the limit value (255 in decimal).
Use a value equal to or less than 255 to write it.

`C1591 (W) assign far pointer to near pointer, bank value ignored`

An attempt is made to assign a far pointer to a near pointer. Only the 2 lower bytes of the far pointer will be used.
Verify the data types near and far.

`C1592 (W) assignment from const pointer to non-const pointer`

A pointer assignment from const to non-const, if attempted, causes the const property to be lost.
Check the description. If correctly written, ignore this warning.

`C1593 (W) assignment from volatile pointer to non-volatile pointer`

A pointer assignment from volatile to non-volatile, if attempted, causes the volatile property to be lost.
Check the description. If correctly written, ignore this warning.

`C1594 (W) far pointer (implicitly) casted by near pointer`

The far pointer has been changed to a near pointer.
Verify the data types near and far.

`C1595 (W) incompatible pointer types`

The type of the object pointed to by a pointer is incompatible with the pointer.
Be sure the object type matches that of the pointer.

`C1596 (W) mismatch function pointer assignment`

The address of a function that has register parameters is assigned to the pointer variable for a function that is not a register parameter type (i.e., not prototyped).
Change the manner in which the pointer variable for the function is declared to the prototype declaration form.

`C1597 (W) RESTRICT qualifier can set only pointer type.`

The RESTRICT qualifier is declared for other than a pointer.
Declare it for only a pointer.

`C1598 (W) near pointer not supported, near qualifier ignored`

A near pointer cannot be used.
Delete the near qualifier.

`C1599 (W) _ext4mptr qualifier can set only pointer type`

The _ext4mptr qualifier is attached to a type that is not a pointer.
To use _ext4mptr, specify a pointer.

`C1600 (W) invalid '%s' operand`

Operations on this type are not permitted under language standards.
Follow the language standard as you write.

`C1611 (W) assignment in comparison statement`

An assignment statement is written in a place where you should write a comparison expression.
You might have written a " = " erroneously whereas it should be " = = ". Check whether it's what you intended.

`C1612 (W) meaningless statement`

The statement terminates with " = =".
You might have written a " = = " erroneously whereas it should be " = ". Check whether it's what you intended.

`C1613 (W) can't get size of function`

A function name is written in the operand of a sizeof operator.

`C1614 (W) can't get size of function, unit size 1 assumed`

The pointer to the function is incremented (++) or decremented (– –). Process will be continued by assuming the increment and decrement value is 1.
Do not increment (++) or decrement (– –) the pointer to a function.

`C1617 (W) cyclic or alarm handler function has argument`

The function specified by #pragma CYCHANDLER or ALMHANDLER is using arguments.
Functions specified by #pragma CYCHANDLER or ALMHANDLER cannot use arguments. Delete the argument.

`C1618 (W) function function -name has no-used argument (variable-name)`

The variable declared in the argument to the function is not used.
Check the variables used.

`C1619 (W) function inlining made dummy return value`

The inline function that should return a value has a return statement that does not return a value.
Change the return statement so that it will return a value.

`C1620 (W) function must be far`

The function is declared with near type.
Declare the function with far type.

`C1621 (W) handler function called`

The function specified by #pragma HANDLER is called.
Be careful not to call a handler function.

`C1622 (W) handler function can't return value`

The function specified by #pragma HANDLER is using a return value.
Functions specified by #pragma HANDLER cannot use a return value. Delete the return value.

`C1623 (W) handler function has argument`

The function specified by #pragma HANDLER is using an argument.
Functions specified by #pragma HANDLER cannot use arguments. Delete the argument.

`C1625 (W) interrupt function called`

The function specified by #pragma INTERRUPT is called.
Be careful not to use an interrupt handling function.

`C1626 (W) interrupt function can't return value`

The interrupt handling function specified by #pragma INTERRUPT is using a return value.
Return values cannot be used in an interrupt handling function. Delete the return value.

`C1627 (W) interrupt function has argument`

The interrupt handling function specified by #pragma INTERRUPT is using an argument.
Arguments cannot be used in an interrupt handling function. Delete the argument.

`C1628 (W) invalid function argument`

The arguments to the function are not written correctly.
Write the arguments to the function correctly.

`C1629 (W) invalid storage class for function, change to extern`

An invalid storage class is used in function declaration. It will be handled as extern when processed.
Change the storage class to extern.

`C1630 (W) non-prototyped function declared`

There is no prototype declaration for the defined function (displayed only when the compile option -Wnon_prototype is specified).
Declare prototype for the function.

`C1631 (W) non-prototyped function used`

A non-prototyped function is called. This error is output only when the compile option -Wnon_prototype is specified.
Write a prototype declaration for the function or do not specify the compile option -Wnon_prototype.

`C1632 (W) old style function declaration`

The function definition is written in format prior to ANSI (ISO) C.
Write the function definition in ANSI (ISO) format.

`C1633 (W) prototype function is defined as nonprototyped function before`

A function, not prototyped before, has prototype for it declared here.
Use the consistent method for declaring functions.

`C1635 (W) register parameter function used before as stack parameter function`

The function having register parameters is used as a function having stack parameters before.
Declare prototype for a function before using it.

`C1636 (W) static variable in inline function`

A declaration of static data is made in the function declared with storage class 'inline'.
Do not declare static data in an inline function.

`C1637 (W) task function called`

The function specified by #pragma TASK is called.
Be careful not to call a task function.

`C1638 (W) task function can't return value`

The function specified by #pragma TASK is using a return value.
Functions specified by #pragma TASK cannot use a return value. Delete the return value.

`C1639 (W) task function has invalid argument`

The function specified by #pragma TASK is using an argument.
Functions specified by #pragma TASK cannot use an argument. Delete the argument.

`C1640 (W) this function used before with non-default argument`

The function after being called is declared as a function that has default arguments.
Declare default arguments before using a function.

`C1641 (W) this interrupt function is called as normal function before`

The function after being called is declared by #pragma INTERRUPT.
Interrupt functions cannot be called. Check the content of #pragma.

`C1642 (W) inline function is called as normal function before, change to static function`

The function after being called is declared as an inline function.
Define an inline function before the first call.

`C1643 (W) xxx was declared but never referenced`

There is a declaration that is not referenced.
Delete the declaration.

`C1644 (W) inline function have invalid argument or return code`

The number of arguments in a call to an inline function does not agree with its prototype declaration.
Make sure the number of arguments in a call to an inline function agrees with its prototype declaration.

`C1645 (W) function 'function -name' size is out of range`

The defined size of the inline function is too large, so that the function cannot be expanded in-line.
Reduce the defined size of the inline function.

`C1671 (W) argument is define by 'typedef', 'typedef' ignored`

Specifier typedef is used in argument declaration. Specifier typedef will be ignored.
Delete typedef.

`C1672 (W) illegal storage class for argument, 'extern' ignore`

An invalid storage class is used in the argument list of function definition.
Specify the correct storage class.

`C1673 (W) enum declared inside parameter list`

The enumerated type declared in a parameter list cannot have its type referenced from outside the function.
Declare the enumerated type outside the function, and not in a parameter list.

`C1674 (W) mismatch prototyped parameter type`

Parameter type is different than that declared in a prototype declaration.
Check the type of parameters.

`C1675 (W) struct declared inside parameter list`

The structure type declared in a parameter list cannot have its type referenced from outside the function.
Declare the structure type outside the function, and not in a parameter list.

`C1676 (W) struct/union/enum declared inside parameter list`

The structure type, union type, and enumerated type declared in a parameter list cannot have their type referenced from outside the function.
Declare these types outside the function, and not in a parameter list.

`C1677 (W) too few parameters`

There are fewer parameters than when declared in a prototype declaration.
Check the number of prototyped parameters.

`C1678 (W) too many parameters`

There are too many parameters than when declared in a prototype declaration.
Check the number of parameters.

`C1679 (W) union declared inside parameter list`

The union type declared in a parameter list cannot have its type referenced from outside the function.
Declare the union type outside the function, and not in a parameter list.

`C1680 (W) uncomplete struct member`

The structure or union members contain incomplete type.
Use the structure and union members that have complete type.

`C1691 (W) 'auto' is illegal storage class`

An invalid storage class is used.
Specify the correct storage class.

`C1692 (W) inline & static conflicted, inline ignored`

Both inline and static are the storage class specifier. They cannot be specified at the same time.
Specify only one of the two at a time.

`C1693 (W) block level extern variable initialize forbid, ignored`

An initialization expression is written in the extern variable declaration of a function.
Delete the initialization expression or change the storage class.

`C1694 (W) external variable initialized, change to public`

An initialization expression is written for the variable declared as extern. Specifier extern will be ignored.
Delete extern.

`C1695 (W) no volatile in previous declaration`

The same declaration already exists, but volatile is nonexistent in the preceding declaration.
Make sure the same variables or functions declared have matching type.

`C1696 (W) no const in previous declaretion`

A function or variable declared without a const qualifier is const-qualified in the definition of the function or variable body.
Make sure const qualification in function and variable declaration and that in the definition of their body are consistent.

`C1697 (W) static declaration of identifier follows non-static`

The same declaration already exists, but static is nonexistent in the preceding declaration.
Make sure the same variables or functions declared have matching storage class.

`C1698 (W) extern/static conflict with previous declaration`

The external/internal linkages differ from the previous declaration. The internal linkage will be assumed.
Do not write multiple declarations that differ in only linkages in the visible scope but have the same name and same type.

`C1711 (W) char array initialized by wchar_t string`

The array of type char is being initialized with a string of type wchar_t.
Make sure the array is initialized with a matching type.

`C1712 (W) size of array shall be a value greater than zero`

The number of array elements is declared by a value equal to or less than 0.
When declaring an array, be sure the number of its elements is equal to or greater than 1.

`C1713 (W) string size bigger than array size`

The size of the initialization expression is greater than that of the variable to be initialized.
Make sure the size of the initialization expression is the same as or smaller than that of the variable.

`C1714 (W) wchar_t array initialized by char string`

The array of type wchar-t is being initialized with a string of type char.
Make sure the array is initialized with a matching type.

`C1716 (W) enumerator value overflow size of unsigned char`

When the compile option -fCE is in use, the enumerator value exceeded 255.
Make sure the enumerator you write is representable by 255 or less.

`C1717 (W) enumerator value overflow size of unsigned int`

The enumerator value exceeded 65,535.
Make sure the enumerator you write is representable by 65,535 or less.

`C1718 (W) enum's bitfield`

The bit-field members are defined using enumerated type.
Use the members of a different type.

`C1719 (W) string terminator not added`

Because the number of array elements and the size of the initialization expression are the same, the '\0' which would otherwise be added at the end of a string will not be added.
Increase the number of array elements.

`C1731 (W) identifier (variable-name) is duplicated`

The variable name is defined twice or more. This declaration will be ignored.
Make sure the variable names are declared only once.

`C1732 (W) identifier (variable-name) is shadowed`

The auto variable that has the same name as the variable name declared for parameter is used. The auto variable will be ignored.
Use any variable name other than those used for parameters.

`C1733 (W) identifier (member-name) is duplicated, this declare ignored`

The member name is defined twice or more. This declaration will be ignored.
Make sure the member names are declared only once.

`C1734 (W) can't get address from register storage class variable`

The & (address) operator is written for the variable of register storage class.
Do not write the & (address) operator for variables of register storage class.

`C1735 (W) No storage class & data type in declare, global storage class & int type assumed`

The variable is declared without storage-class and type specifiers. It will be processed as int.
Write the storage-class and type specifiers.

`C1736 (W) 'register' is illegal storage class`

An invalid storage class is used.
Specify the correct storage class.

`C1737 (W) near/far is conflict beyond over typedef`

The type defined by specifying near/far is again defined by specifying near/far when referencing it.
Write the type specifier correctly.

`C1754 (W) invalid return type`

The expression of the return statement does not match the type of the function.
Make sure that the return value is matched to the type of the function or that the type of the function is matched to the return value.

`C1755 (W) redefined type`

The type name already defined with typedef is redefined.
Use another type name or check whether the type name is erroneously written.

`C1756 (W) redefined type name of (identifier)`

The same identifier is defined twice or more by typedef.
Write the identifier correctly.

`C1800 (W) section name 'interrupt' no more used`

The section name specified by pragma SECTION uses 'interrupt'.
A section name 'interrupt' cannot be used. Change it to another.

`C1803 (W) the same identifier is stored in a different section, previous section is used`

The same variable or function declared multiple times has a different section location specified by #pragma SECTION.
For the same variable or function declared, specify the same section.

`C1814 (W) non initialized variable 'variable-name' is used`

An uninitialized auto variable is being referenced.
Set a value for the variable before referencing it.

`C1831 (W) case value is out of range`

The case value exceeds the range representable by an expression for branch condition of a switch statement.
Make sure the case value does not exceed the range of the switch parameter.

`C1832 (W) compile option -fauto_over_255 is specified, #pragma SBDATA was ignored`

When the option -fauto_over_255 is specified, #pragma SBDATA cannot be specified.
Specify either one of the two.

`C1833 (W) init elements overflow, ignored`

The initialization expressions exceeded the size of the variable to be initialized.
Make sure the number of initialization expressions does not exceed the size of the variables to be initialized.

`C1834 (W) keyword (keyword) are reserved for future`

A keyword reserved for use in the future is used.
Change it to a different name.

`C1835 (W) large type was implicitly cast to small type`

The upper bytes (word) of value may be lost by an assignment from large type to smaller type.
Check the type. If the description is correct, ignore this warning.

`C1836 (W) No initialized of variable-name`

It is probable that the register variables are used without being initialized.
Make sure the register variables are assigned the initial value.

`C1837 (W) no restrict in previous declaration`

The same declaration already exists, but restrict is nonexistent in the preceding declaration.
Make sure the same variables or functions declared have matching type.

`C1838 (W) overflow in floating value converting to integer`

A very large floating-point value that cannot be stored in integer type is being assigned to integer type.
Reexamine the assignment expression.

`C1839 (W) standard library "function -name()" need "include file name"`

The standard library function is used without its header file included.
Be sure to include the header file.

`C1840 (W) this feature not supported now, ignored`

This is a syntax error. Do not use this syntax because it is reserved for future extension.
Write the description correctly.

`C1841 (W) underflow in floating value converting to integer`

A floating-point constant of a large size not representable by integer type is being converted to integer type.
Make sure the values you use are in the range representable by integer type to which converted.

`C1842 (W) zero divide in constant folding`

The divisor in the division operator or remainder operator is 0.
Use any value other than 0 for the divisor.

`C1843 (W) zero divide, ignored`

The divisor in the division operator or remainder operator is 0.
Use any value other than 0 for the divisor.

`C1844 (W) zero width for bitfield`

The bit-field width is 0.
Write a bit-field equal to or greater than 1 in width.

`C1847 (W) no _ext4mptr is previous declaration`

The same declaration already exists, but _ext4mptr is nonexistent in the preceding declaration.
Make sure the same variables or functions declared have matching type.

`C1848 (W) meaningless statements deleted in optimize phase`

Meaningless statements were deleted by optimization.
Delete meaningless statements.

`C1849 (W) this comparison is always true`

Comparison is made that always results in true.
Check the conditional expression.

`C1850 (W) this comparison is always false`

Comparison is made that always results in false.
Check the conditional expression.

`C1851 (W) compile option -fSB_auto(-fSBA) is specified, #pragma SBDATA was ignored`

The option -fSB_auto and #pragma SBDATA cannot be used at the same time.
Specify either one of the two.

`C1860 (W) -OR, -OS duplicated option`

-OR and -OS cannot be used at the same time.
Specify either one of the two.

`C1861 (W) Option name A, option name B duplicated option, option name C is ignore`

The option name A and option name B cannot be used at the same time. Option name C will be ignored.
Specify either one of the two.

`C1862 (W) Can't use option nameA with option name B, option name A is ignored.`

The option name A and option name B cannot be used at the same time. Option name A will be ignored.
Specify either one of the two.

`C1863 (W) Invalid option-name value (value)`

The value set for the option name is invalid.
Set the correct value.

`C1864 (W) Unknown option type option (option-name)`

The option (option name) does not exist.
Use the correct option name.

`C1865 (W) Unknown option (option-name)`

The option (option name) does not exist.
Use the correct option name.

`C2004 (E) can't open command file`

The command file specified by @ cannot be opened.
Specify the correct file name.

`C2005 (E) command-file line characters exceed 2048`

The number of characters per line in the command file exceeds 2,048.
Make sure the number of characters per line in the command file is 2,048 or less.

`C2008 (E) Invalid suffix '.xxx'`

A file extension unrecognizable by NC30 (one other than .c, .cpp, .cc, .cp, .i, .a30, and .obj) is used.
Use the correct extension to specify a file.

`C2009 (E) Invalid option '-?'`

An invalid compile option -? is specified. Or the compile option -? does not have the necessary parameter.
Check whether the compile option -? is correct. Or specify the necessary parameter following the compile option -?.

`C2010 (E) Too many command files`

The @ command file is specified twice or more.
Make sure the @ command file is specified only once.

`C2011 (E) too many options`

The number of specified compile options is 100 or more.
Specifiable compile options are limited to 99 occurrences.

`C2012 (E) -r8ce, -r8c duplicated option`

The -R8C option and -R8CE option are specified at the same time.
Do not specify the -R8C option and -R8CE option at the same time.

`C2013 (E) Can't specify twice option 'option'`

The same option is specified twice or more, or conflicting options are specified at the same time.
Make sure the option you specify is specified only once. Also, specify either one of the conflicting options.

`C2014 (E) Can't specified 'option' with -S option`

An option not specifiable simultaneously with the -S option is specified.
Do not specify this option and the -S option at the same time.

`C2015 (E) Invalid NCKIN value 'xxxx'`

The environment variable NCKIN has an invalid value set in it.
Make sure the value set in the environment variable NCKIN is either SJIS or EUC.

`C2017 (E) Illegal option 'option' can't specify together with -lang=ecpp option`

An option not specifiable simultaneously with -lang=ecpp is specified.
Do not specify this option and -lang=ecpp at the same time.

`C2018 (E) Illegal option 'option' can't specify together with -rtti,-exception,-template option`

The -lang=ecpp option is specified simultaneously with -rtti=on or -exception.
Do not specify the -lang=ecpp option and -rtti=on or -exception at the same time.

`C2024 (E) Can't be specified to a file name`

The -o option has a string beginning with a hyphen (-) specified in its parameter.
For the parameter (file) of the -o option, specify other than the one that begins with a hyphen (-).

`C2026 (E) Can't specify 'option A' with 'option B' option`

The option A you've specified cannot be specified along with option B.
Do not specify option A and option B at the same time.

`C2029 (E) No directory 'directory', environment variable 'environment variable-name'`

The directory set in the environment variable cannot be found.
Check whether the directory set in the environment variable is correct.

`C2500 (E) Sorry, compilation terminated because of too many errors`

Errors in the source file exceeded the upper limit (50 occurrences).
Correct the errors detected before this message is output.

`C2501 (E) Sorry, compilation terminated because of these errors in function-name`

An error occurred in the function indicated by a function name. Compilation will be terminated.
Correct the errors detected before this message is output.

`C2502 (E) can't read C source from filename line number for error message`

The source line in error cannot be displayed. The file indicated by filename cannot be found or the line number does not exist in the file.
Check whether the file actually exists.

`C2504 (E) can't open C source filename for error message`

The source line in error cannot be opened.
Check whether the file actually exists.

`C2505 (E) Sorry stack frame memory exhaust, max 'the maximum total size of arguments' bytes(argument) but now the current`

total size of arguments' bytes.
The total size of arguments passed via stack is too large.
Reduce the size to within the maximum value displayed.

C2506 (E) Sorry stack frame memory exhaust, max 'stack size which can be used by a function' bytes(auto) but now the current

total size' bytes.

The total size of arguments passed via stack and auto variables is too large.

Reduce the size to within the maximum value displayed.

C2508 (E) can't refer to the range outside of the stack frame

A location outside the stack frame area is being referenced.

Specify correctly.

C2509 (E) too many operators

There are too many operators in one line.

Limit the number of operators in one line to less than 1,000.

C2512 (E) #pragma pragma-name & function prototype mismatched

The function specified with #pragma pragma name and the contents of parameters in its prototype declaration do not agree.

Make sure the parameters in a function prototype declaration agree with the specified function.

C2514 (E) Invalid #pragma OS extended function interrupt number

The INT number written in the #pragma OS extension feature cannot be specified.

Specify correctly.

C2514 (E) Invalid #pragma INTCALL interrupt number

The INT number written in #pragma INTCALL cannot be specified.

Specify correctly.

C2515 (E) Invalid #pragma SPECIAL special page number

The number or format specification written in #pragma SPECIAL is incorrect.

Specify correctly.

C2516 (E) Invalid #pragma INTERRUPT vector number

The number or format specification written in #pragma INTERRUPT is incorrect.

Specify correctly.

C2518 (E) multiple #pragma EXT4MPTR's pointer, ignored

More than one #pragma EXT4MPTR is declared.

Do not specify more than one #pragma EXT4MPTR.

C2519 (E) asm()'s string must have 1 $$

This asm function must have at least one $$.

Use one $$.

C2520 (E) asm()'s string must have 1 $$ or $@

This asm function must have at least one $$ or $@.

Use one $$ or $@.

C2521 (E) asm()'s string must have 1 $@

This asm function must have at lease one $@.

Use one $@.

`C2522 (E) asm()'s string must have only 1 $b`

In an asm statement, $b can be written only once.
Make sure $b is written only once.

`C2523 (E) asm()'s string must not have more than 3 $$ or $@`

In an asm statement, $$ or $@ is written three times or more.
Make sure $$ ($@) is written twice or less.

`C2525 (E) floating type's bitfield`

A bit-field of invalid type is declared.
Use integer type for bit-fields.

`C2525 (E) invalid asm()'s argument`

The variables usable in an asm statement are auto variables and arguments.
Use auto variables or arguments to write an asm statement.

`C2526 (E) #pragma PARAMETER functions register not allocated`

A register indicated in the function that is specified by #pragma PARAMETER cannot be written.
Write a register correctly.

`C2527 (E) #pragma pragma-name's function must be declared before use, #pragma is ignored`

#pragma is specified after a function call.
Specify #pragma before calling the target function.

`C2528 (E) #pragma BITADDRESS variable is not _Bool type`

The variable specified by #pragma BITADDRESS is not _Bool type.
Be sure the variables specified by #pragma BITADDRESS are _Bool type.

`C2529 (E) #pragma pragma-name format error, ignored`

The content following #pragma pragma name is incorrect.
Write it in the correct format.

`C2531 (E) #pragma INTCALL function's argument on stack`

Whereas the body of a function declared by #pragma INTCALL is written in C, the arguments are passed via stack.
When writing the body of a function declared by #pragma INTCALL in C, specify a type for which the arguments are passed via register.

`C2532 (E) #pragma pragma-name function argument is long-long or double`

Type long long or type double is used for the arguments to the function specified by #pragma pragma name.
For the functions specified by "#pragma pragma name function name", type long long and type double cannot be specified. Use other types.

`C2533 (E) #pragma pragma-name function argument is struct or union`

In a prototype declaration for the function specified with #pragma pragma name, struct or union type is specified.
In a prototype declaration, specify int or short type, a pointer type in size of 2 bytes, or an enumerated type.

`C2534 (E) #pragma pragma-name must be declared before use`

The definition of a function specified by #pragma pragma name is written after a call to that function.
Declare it before calling the function.

`C2535 (E) #pragma pragma-name function-name redefined`

The same function is defined twice or more in #pragma pragma name.
Make sure #pragma pragma name is declared only once.

`C2537 (E) #pragma pragma-name function must be prototyped`

The function specified with #pragma pragma name is called while there is no prototype declaration for it.
Make sure the function specified with #pragma pragma name has its prototype declared before a call.

`C2551 (E) mismatch prototyped parameter type`

Parameter type is different than that declared in a function prototype declaration.
Check the parameter type.

`C2552 (E) 'function-name' function has struct argument`

An inline function cannot have a structure as the argument to it.
Make sure the functions that have arguments of structure type are not used as inline functions.

`C2554 (E) 'function-name' is recursion, a function of recursive call can not be described inline qualifier`

Inline functions cannot be called recursively.
Eliminate inline specification from the functions that are recursively called.

`C2555 (E) can't get inline function's address by '&' operator`

A & operator is written in an inline function.
Do not write a & operator in inline functions.

`C2556 (E) conflict function argument type of function-name`

The argument list contains variables that have the same name.
Change the variable names.

`C2557 (E) declared register parameter function's body declared`

The function declared by #pragma PARAMETER has its body defined in C.
For functions declared by #pragma PARAMETER, do not write the function body in C.

`C2558 (E) function initialized`

An initialization expression is written for function declaration.
Remove the initialization expression.

`C2559 (E) function member declared`

Structure or union members are used to specify function type.
Write the members correctly.

`C2560 (E) function returning a function declared`

The type of return value in function declaration is a function type.
Change the type of return value to a pointer to function or other type.

`C2561 (E) function returning an array`

The type of return value in function declaration is an array type.
Change the type of return value to a pointer to function or other type.

`C2562 (E) handler function called`

The function specified by #pragma HANDLER is called.
Be careful not to call a handler function.

`C2563 (E) default function argument conflict`

In a function prototype declaration, the default value of a parameter is declared twice or more.
Make sure the default value of an argument is declared only once.

`C2564 (E) inline function have invalid argument or return code`

The inline function contains an invalid argument or invalid return value.
Specify the correct argument or return value.

`C2565 (E) inline function is called as normal function before`

The inline function is called before declaration as an ordinary function.
Check the function.

`C2566 (E) inline function's address used`

The address of an inline function is being referenced.
Do not use the address of an inline function.

`C2567 (E) inline function's body is not declared previously`

The body of the inline function is not defined.
When using an inline function, define the function body prior to a function call.

`C2568 (E) interrupt function called`

The function specified by #pragma INTERRUPT is called.
Do not call an interrupt handling function.

`C2569 (E) invalid function argument`

In argument declaration of the function definition, an argument not included in the argument list is declared.
Declare arguments that are included in the argument list.

`C2570 (E) invalid function declare`

The function definition contains an error.
Check the line in error or the function definition immediately preceding it.

`C2571 (E) invalid function default argument`

The default argument of the function is incorrect.
This error occurs when the prototype declaration for a function that has default parameters and the parameters in its definition do not agree. When writing a prototype declaration for a function and its definition, be sure that they agree.

`C2572 (E) invalid function[] operand`

Arrays of function type cannot be used.
Use an array of function pointers.

`C2573 (E) invalid function's argument declaration`

The declaration of the function arguments contains an error.
Write the declaration correctly.

`C2574 (E) redefine function function-name`

The function indicated by function name is defined twice or more.
The function can be defined only once. Make sure there is only one definition of the function.

`C2575 (E) return expression is in void type function`

The function definition that returns void contains a return statement that returns a value.
Make sure a return statement in such a function definition does not return a value.

`C2576 (E) task function called`

The function specified by #pragma TASK cannot be called in the same way as for ordinary functions.
For details on how to call a function specified by #pragma TASK, refer to the RTOS manual.

`C2577 (E) unknown function argument variable-name`

An argument not included in the argument list is specified.
Check the argument.

`C2591 (E) array of functions declared`

In the array declaration, an array of functions themselves, not an array of pointers to the functions, is declared.
Change it to a pointer array to functions, etc.

`C2592 (E) array size is not constant integer`

The number of elements in the array declaration is not a constant.
Use a constant to write the number of elements.

`C2593 (E) incomplete array access`

A multi-dimensional array of incomplete type is being referenced.
Explicitly specify the size of the multi-dimensional array.

`C2594 (E) invalid initializer on array`

The initialization expression contains an error.
Check to see if the number of initialization expressions in the parentheses matches the number of array elements and the number of structure members.

`C2595 (E) invalid initializer on char array`

The initialization expression contains an error.
Check to see if the number of initialization expressions in the parentheses matches the number of array elements and the number of structure members.

`C2596 (E) size of incomplete array type`

An attempt is made to find sizeof of an array of unknown size. This is an invalid size.
Specify the size of the array.

`C2597 (E) size of uncomplete type's array`

The size of an incomplete array cannot be obtained.
If it is necessary to get array size, change the array type to complete type.

`C2598 (E) too large array size : number of bytes`

The array size is excessively large.
Reduce the array size.

`C2599 (E) uncomplete array pointer operation`

An attempt is made to reference an array of incomplete type via pointer.
Define a complete array first.

`C2600 (E) void array is invalid type, int array assumed`

An array of void type cannot be declared. The compiler will continue processing the array assuming it to be an int-type array.
Write the type specifier correctly.

`C2601 (E) zero size array member`

An array whose size is zero.
Specify the size clearly.
The structure members include an array whose size is zero.
Arrays of size 0 cannot be a structure member.

`C2603 (E) incomplete struct get by [ ]`

An array of (incomplete) structures or unions that do not have valid members is being referenced or initialized.
Define complete structures or unions first.

`C2604 (E) incomplete struct initialized`

An (incomplete) structure or union that does not have valid members is being initialized.
Define a complete structure or union first.

`C2605 (E) incomplete struct return function call`

A function that has as its return value the type of (incomplete) structure or union that does not have valid members is called.
Define a complete structure or union first.

`C2606 (E) incomplete struct/union(tag-name)'s member access`

Members of an (incomplete) structure or union that does not have valid members are being referenced.
Define a complete structure or union first.

`C2607 (E) incomplete struct/union's member access`

Members of an (incomplete) structure or union that does not have valid members are being referenced.
Define a complete structure or union first.

`C2608 (E) invalid initializer on struct`

The initialization expression contains an error.
Check to see if the number of initialization expressions in the parentheses matches the number of array elements and the number of structure members.

`C2609 (E) invalid struct or union type`

Structure or union members are referenced for the data of enumerated type .
Write it correctly.

`C2610 (E) not struct or union type`

The left-side expression of -> is not structure or union type.
Use structure or union type to write the left-side expression of ->.

`C2611 (E) redefinition tag of struct tag-name`

The structure is defined twice.
Make sure the structure is defined only once.

`C2612 (E) struct or enum's tag used for union`

The tag name of structure or enumerated type is used as the tag name of a union.
Change the tag name.

`C2613 (E) struct or union's tag used for enum`

The tag name of a structure or union is used as a tag name of enumerated type.
Change the tag name.

`C2614 (E) union or enum's tag used for struct`

The tag name of structure or enumerated type is used as the tag name of a structure.
Change the tag name.

`C2615 (E) unknown pointer to structure idetifier "variable-name"`

The left-side expression of -> is not structure or union type.
Use structure or union type to write the left-side expression of ->.

`C2616 (E) unknown size of struct or union`

An incomplete structure or union which has its size not determined is used.
Before declaring the variables of a structure or union, declare the structure or union first.

`C2617 (E) unknown structure idetifier "variable-name"`

The left-side expression of . is not .structure or union type.
Use structure or union type to write a left-side expression of ..

`C2618 (E) redefinition tag of union tag-name`

The union is defined twice.
Make sure the union is defined only once.

`C2619 (E) invalid enumerator initialized`

The initial value of the enumerator is erroneously specified by writing a variable name, for example.
Write the initial value of the enumerator correctly.

`C2620 (E) redefinition tag of enum tag-name`

The enumerator is defined twice.
Make sure the enumerator is defined only once.

`C2621 (E) bitfield width exceeded`

The bit-field width exceeds the bit width of data type.
Make sure the bit-field you write is within the bit width of the declared data type.

`C2622 (E) bitfield width is not constant integer`

The bit width of the bit-field is not a constant.
Use a constant to write the bit width.

`C2623 (E) can't get bitfield address by '&' operator`

The & operator is written for the bit-field type.
Do not write the & operator for the bit-field type.

`C2624 (E) can't get size of bitfield`

An attempt is made to obtain the size of a bit-field.
The size of a bit-field cannot be obtained.

`C2626 (E) invalid bitfield declare`

The bit-field declaration contains an error.
Write it correctly.

`C2627 (E) invalid size of bitfield`

An attempt is made to obtain the size of a bit-field.
Do not write a bit-field in this declaration.

`C2628 (E) invalid type's bitfield`

A bit-field of invalid type is declared.
Use integer type for bit-fields.

`C2629 (E) long long type's bitfield`

A bit-field of long long type is written.
Note that long long type cannot be declared for bit-fields. Use another type to declare a bit-field.

`C2630 (E) invalid array type`

An array of invalid type cannot be declared.
When declaring a multi-dimensional array, be sure to specify the number of array elements.

`C2651 (E) not static initializer for variable-name`

The initialization expression for static variables is erroneous. For example, it may be written in the form of a function call.
Write the initialization expression correctly.

`C2652 (E) 'static' is illegal storage class for argument`

In argument declaration, an inappropriate storage class is used.
Use the correct storage class.

`C2661 (E) do while(void) statement`

Type void is used for the expression of a do-while statement.
Write scalar type for the expression of a do-while statement.

`C2662 (E) do while(struct/union) statement`

Type struct or union is used for the expression of a do-while statement.
Write scalar type for the expression of a do-while statement.

`C2663 (E) for(; struct/union;) statement`

Type struct or union is used for the second expression of a for statement.
Write scalar type for the second expression of a for statement.

`C2664 (E) if(struct/union) statement`

Type struct or union is used for the expression of an if statement.
Write scalar type for the expression of an if statement.

`C2665 (E) if(void) statement`

Type void is used for the expression of an if statement.
Write scalar type for the expression of an if statement.

`C2666 (E) invalid break statements`

The break statement is used where it cannot be written.
Write it in switch, while, do-while, or for.

`C2667 (E) invalid case statements`

The case statement is written in other than a switch statement.
Do not write it in other than a switch statement.

`C2668 (E) invalid continue statements`

The continue statement is used where it cannot be written.
Write it in while, do-while, or for.

`C2669 (E) invalid default statements`

The switch statement contains an error.
Write the switch statement correctly.

`C2670 (E) invalid switch statement`

The switch statement contains an error.
Write it correctly.

`C2671 (E) while(struct/union) statement`

Type struct or union is used for the expression of a while statement.
Write scalar type for the expression of a while statement.

`C2672 (E) while(void) statement`

Type void is used for the expression of a while statement.
Write scalar type for the expression of a while statement.

`C2673 (E) for(; void ;) statement`

Type void is used for the second expression of a for statement.
Write scalar type for the second expression of a for statement.

`C2691 (E) auto variable's size is zero`

An array whose number of elements is zero or an array that has no element number is declared in the auto area.
Declare it correctly.

`C2692 (E) invalid environment variable : environment variable-name`

The variable name specified by environment variable NCKIN/NCKOUT is not SJIS or EUC.
Check the environment variable.

`C2693 (E) unknown variable variable-name used`

An undefined variable name is used.
Define the variable.

`C2694 (E) unknown variable variable-name`

An undefined variable name is used.
Define the variable.

`C2695 (E) unknown variable "variable-name" used in asm()`

An undefined variable name is used in the asm statement.
Define the variable.

`C2696 (E) can't get void value`

An attempt is made to reference the value of void type in an expression.
Check the data type.

`C2697 (E) case value is duplicated`

The case value is used more than once.
Make sure the case value that you used once is not used again within one switch statement.

`C2698 (E) floating point value overflow`

The value of the floating-type constant exceeds the representable range.
Make sure the constant value is within the range.

`C2699 (E) invalid case value`

The case value is erroneous.
Write a value of integer type or enumerated type.

`C2701 (E) void value can't return`

The value cast to type void is used for the return value of the function.
Write correctly.

`C2702 (E) argument type given both places`

In argument declaration of the function definition, an argument declared once in the argument list is declared here again.
Declare the argument in either the argument list or argument declaration.

`C2705 (E) illegal storage class for argument, 'interrupt' ignored`

An interrupt function is declared in declaration statement within the function.
Declare it outside the function.

`C2706 (E) invalid lvalue`

The left side of the assignment expression is not substitutable.
Write a substitutable object on the left side of the expression.

`C2707 (E) can't set argument`

Because the prototype declaration for a function and the type of an argument to the function do not match, the argument cannot be set in a register (parameter).
Correct mismatch of the type.

`C2708 (E) illegal storage class for argument, 'inline' ignored`

An inline function is declared in declaration statement within the function.
Declare it outside the function.

`C2721 (E) switch's condition is floating`

Floating type is used in the expression of a switch statement.
Use integer type or enumerated type.

`C2722 (E) switch's condition is void`

void type is used in the expression of a switch statement.
Use integer type or enumerated type.

`C2723 (E) switch's condition must integer`

Invalid types other than integer and enumerated types are used for the expression of a switch statement.
Use integer type or enumerated type.

`C2743 (E) 'const' is duplicate`

const is written more than once.
Write the type qualifier correctly.

`C2744 (E) default: is duplicated`

The default value is used twice or more.
Two or more default labels are used in one switch statement.
Make sure the default label is used only once in one switch statement. (Not including default labels in nested switch statements)

`C2745 (E) identifier (variable-name) is duplicated`

The variable is defined twice or more.
Specify the variable definition correctly.

`C2746 (E) 'restrict' is duplicate`

The restrict qualifier in declaration is duplicated.
Declare only once for one target of qualification.

`C2747 (E) 'volatile' is duplicate`

volatile is written more than once.
Write the type qualifier correctly.

`C2748 (E) '_ext4mptr' is duplicated`

_ext4mptr is written repeatedly.
Delete duplicates until there is only one _ext4mptr.

`C2761 (E) conflict declare of variable-name`

The variable is defined twice with different storage classes each time.
Use the same storage class to declare a variable twice.

`C2763 (E) duplicate frame position defind variable-name`

auto variables with the same identifier are written more than once.
Write correctly.

`C2764 (E) Empty declare`

Only storage class and type specifiers are found.
Write a declarator.

`C2765 (E) 'far' & 'near' conflict`

The near and far declarations for the same variable (function) do not match.
Write near and far correctly.

`C2766 (E) parse error at near 'character string'`

A noninterpretable string is found.
Rewrite it so that it conforms to C/C++ syntax.

`C2767 (E) parse error at near`

A noninterpretable string is found.
Rewrite it so that it conforms to C/C++ syntax.

`C2780 (E) redeclare of variable or enumerator`

The variable name or enumerator is defined twice or more.
Change either of the duplicate variable names.

`C2781 (E) invalid lvalue at '=' operator`

The left side of the assignment expression is not substitutable.
Write a substitutable object on the left side of the expression.

`C2782 (E) invalid ' ? : ' operand`

The ?: operator is written erroneously.
Check each expression of the operator. Also, make sure the types of expressions on the left and right sides of : are compatible type.

`C2782 (E) invalid '!=' operands`

The != operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '&&' operands`

The && operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '&' operands`

The & operator is written erroneously.
Check the expression to the right of the operator.

`C2782 (E) invalid '&=' operands`

The &= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '()' operand`

The left-side expression of ( ) is not a function.
Write a function or a pointer to function for the left-side expression of ( ).

`C2782 (E) invalid '*' operands`

If multiplication, the * operator contains an error. If * is a pointer operator, the right-side expression is not pointer type.
For a multiplication, check the expressions on the left and right sides of the operator. For a pointer, check the type of the right-side expression.

`C2782 (E) invalid '*=' operands`

The *= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '+' operands`

The + operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '+=' operands`

The += operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '-' operands`

The - operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '-=' operands`

The -= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '/=' operands`

The /= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '<<' operands`

The << operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '<<=' operands`

The <<= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '<=' operands`

The <= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '=' operand`

The = operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '= =' operands`

The = = operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '>=' operands`

The >= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '>>' operands`

The >> operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '>>=' operands`

The >>= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '[ ]' operands`

The left-side expression of [ ] is not an array or pointer type.
Write an array or pointer type for the left-side expression of [ ].

`C2782 (E) invalid '^=' operands`

The ^= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '¦=' operands`

The ¦= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '¦¦' operands`

The ¦¦ operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid '%=' operands`

The %= operator is written erroneously.
Check the expressions on the left and right sides of the operator.

`C2782 (E) invalid ++ operands`

The ++ unary operator or postfix operator is erroneously written.
For the unary operator, check the right-side expression. For the postfix operator, check the left-side expression.

`C2782 (E) invalid -- operands`

The -- unary operator or postfix operator is erroneously written.
For the unary operator, check the right-side expression. For the postfix operator, check the left-side expression.

`C2782 (E) invalid (? ;)'s condition`

The ternary operator is erroneously written.
Check the ternary operator.

`C2782 (E) invalid CAST operand`

The cast operator contains an error. The void type cannot be cast to any other type; it can neither be cast from a structure or union nor can it be cast to other structure or union..
Write the expression correctly.

`C2784 (E) invalid unary '!' operands`

The ! unary operator is erroneously written.
Check the right-side expression of the operator.

`C2784 (E) invalid unary '+' operands`

The + unary operator is erroneously written.
Check the right-side expression of the operator.

`C2784 (E) invalid unary '-' operands`

The - unary operator is erroneously written.
Check the right-side expression of the operator.

`C2784 (E) invalid unary '~' operands`

The ~ unary operator is erroneously written.
Check the right-side expression of the operator.

`C2785 (E) invalid cast operator`

The cast operator is erroneously written.
Write it correctly.

`C2786 (E) invalid (? :)'s condition`

The conditional expression of the condition operator (? :) is invalid.
Write the conditional expression correctly.

`C2787 (E) invalid -> used`

The left-side expression of -> is not a pointer type to structure or union.
Use a pointer type to structure or union to write the left-side expression.

`C2788 (E) invalid operation for pointer to incomplete type`

Invalid operation is performed on pointer to incomplete type.
Define structure members or specify the number of array elements to make the subject complete.

`C2789 (E) can't get address from register storage class variable`

The address of a register variable cannot be obtained.
If it is necessary to get address, remove the register qualification.

`C2801 (E) invalid redefined type name of (identifier)`

The same identifier name is defined by typedef more than once.
Write the identifier name correctly.

`C2802 (E) invalid return type`

The return value of the function is incorrect.
Write it correctly.

`C2803 (E) invalid type specifier`

The same type specifier is written more than once as in "int int i;" or an incompatible type specifier is written as in "float int i;".
Write the type specifier correctly.

`C2804 (E) invalid type specifier, long long long`

Type specifier 'long' is written thrice or more in type declaration.
Check the type declaration.

`C2805 (E) invalid void type, int assumed`

A variable of void type cannot be declared. The compiler will continue processing assuming it to be int type.
Write the type specifier correctly.

`C2807 (E) type redeclaration of variable-name`

The variable is defined twice with different types each time.
Use the same type to declare a variable twice.

`C2808 (E) too many storage class of typedef`

A storage class specifier such as extern, typedef, static, auto, or register is written more than once in declaration.
Do not write a storage class specifier more than once.

`C2809 (E) typedef initialized`

An initialization expression is written for the variable declared by typedef.
Delete the initialization expression.

`C2821 (E) invalid initializer`

The initialization expression contains an error. For example, there are too many parentheses, there are many initialization expressions, a static variable in the function is initialized by an auto variable, or a variable is initialized by another variable
Write the initialization expression correctly.

`C2822 (E) invalid initializer of variable-name`

The initialization expression contains an error. For example, a variable is written for the initialization expression of a bit-field.
Write the initialization expression correctly.

`C2823 (E) invalid initializer on scalar`

The initialization expression contains an error.
Check to see if the number of initialization expressions in the parentheses matches the number of array elements and the number of structure members.

`C2824 (E) invalid initializer, too many brace`

Too many braces { } are used in a scalar-type initialization expression of auto storage class.
Reduce the number of braces { } used.

`C2825 (E) invalid member`

The member reference is erroneously written.
Write it correctly.

`C2826 (E) invalid member used`

The member reference is erroneously written.
Write it correctly.

`C2827 (E) invalid push`

Type void is pushed in function argument, etc.
Type void cannot be pushed.

`C2828 (E) invalid strage class for data`

The storage class is erroneously specified.
Write it correctly.

`C2829 (E) invalid truth expression`

The void, struct, or union type is used in the first expression of a conditional expression (?:).
Use scalar type to write this expression.

`C2830 (E) label label redefine`

The same label is defined twice in one function.
Change the name of either label.

`C2834 (E) size of incomplete type`

An undefined structure or union is written in the operand of the sizeof operator.
Define the structure or union first.
The number of elements of an array defined in the operand of the sizeof operator is unknown.
Specify the number of elements in an array when declaring it.

`C2835 (E) No declarator`

The declaration statement is incomplete.
Write a complete declaration statement.

`C2836 (E) reinitialized of variable-name`

An initialization expression is specified twice for the same variable.
Specify the initialization expression only once.

`C2851 (E) size of void`

An attempt is made to obtain the size of void. This is an invalid size.
The size of void cannot be obtained.

`C2852 (E) too big address`

An attempt is made to set an address in size of 32 bits or more.
Make sure the set values fit in the address range of the microprocessor used.

`C2853 (E) too big data-length`

An attempt is made to set an address in size of 32 bits or more.
Make sure the set values fit in the address range of the microprocessor used.

`C2854 (E) undefined label "label" used`

The jump-address label for goto is not defined in the function.
Define the jump-address label in the function.

`C2855 (E) unknown member member-name used`

A member not registered in structure or union members is being referenced.
Check the member name.

`C2856 (E) syntax error`

This is a syntax error.
Write correctly.

`C3001 (F) Arg list too long`

The command line entered when starting each implementation exceeds the number of characters defined by the system.
Specify a compile option to ensure that the number of characters defined by the system is not exceeded. Use the compile option -v to check the command line of each implementation

`C3002 (F) Permission denied`

Unable to execute each implementation.

Check access rights to each implementation. Or, if permission is OK, check whether the directory of each implementation is correctly set in the environment variable.

`C3003 (F) Invalid argument`

This is an internal error (which does not normally occur).

Please contact Renesas.

`C3004 (F) Too many open files`

This is an internal error (which does not normally occur).

Please contact Renesas.

`C3005 (F) No such file or directory`

Unable to execute each implementation.

Check whether the directory of each implementation is correctly set in the environment variable.

`C3006 (F) Exec format error`

The executable file of each implementation is corrupted.

Please reinstall.

`C3007 (F) Not enough core`

The swap area is insufficient.

Increase the swap area.

`C3008 (F) Result too large`

This is an internal error (which does not normally occur).

Please contact Renesas.

`C3010 (F) Cannot analyze error`

This is an internal error (which does not normally occur).

Please contact Renesas.

`C3012 (F) Can't get environment variable(environment variable-name)`

The environment variable has no values specified. Or the value is invalid.

Set a value of the environment variable.

`C3013 (F) Core dump(command_name)`

The implementation caused a core dump. Enclosed in parentheses is the implementation that caused the core dump.

Each implementation is not executed correctly. Check the environment variable or the directory that contains each implementation. If the implementation still does not run correctly, please contact Renesas.

`C3014 (F) Can't create temporary file`

Failed to open a temporary file.

Check the disk capacity or system status.

`C3507 (F) can't open file name`

Unable to open a file.

Check permission to the file.

`C3508 (F) can't output to file name`

Unable to write to the file.
Check the remaining space of the disk or access rights to the file.

`C3514 (F) No #pragma ENDASM`

There is no matching #pragma ENDASM for #pragma ASM.
Write #pragma ENDASM.

`C3517 (F) Not enough memory`

The memory space is insufficient.
Increase the memory space or the virtual memory of Windows.

`C4000-C4999 (-) Internal error`

An internal error occurred during compilation.
Report the error occurrence to your local Renesas sales office.

`C5001 (E) Last line of file ends without a newline`

`C5002 (E) Last line of file ends with a backslash`

`C5003 (F) #include file "file name" includes itself`

`C5004 (F) Out of memory`

`C5005 (F) Could not open source file "name"`

`C5006 (E) Comment unclosed at end of file`

`C5007 (E) (I) Unrecognized token`

`C5008 (E) (I) Missing closing quote`

`C5009 (I) Nested comment is not allowed`

`C5010 (E) "#" not expected here`

`C5011 (E) (W) Unrecognized preprocessing directive`

`C5012 (E) (W) Parsing restarts here after previous syntax error`

`C5013 (E) (F) Expected a file name`

`C5014 (E) Extra text after expected end of preprocessing directive`

`C5016 (F) "name" is not a valid source file name`

`C5017 (E) Expected a "]"`

`C5018 (E) Expected a ")"`

C5019 (E) Extra text after expected end of number

C5020 (E) Identifier "name" is undefined

C5021 (W) Type qualifiers are meaningless in this declaration

C5022 (E) Invalid hexadecimal number

C5023 (E) Integer constant is too large

C5024 (E) Invalid octal digit

C5025 (E) Quoted string should contain at least one character

C5026 (E) Too many characters in character constant

C5027 (W) Character value is out of range

C5028 (E) Expression must have a constant value

C5029 (E) Expected an expression

C5030 (E) Floating constant is out of range

C5031 (E) (W) Expression must have integral type

C5032 (E) Expression must have arithmetic type

C5033 (E) Expected a line number

C5034 (E) Invalid line number

C5035 (F) #error directive: "line number"

C5036 (E) The #if for this directive is missing

C5037 (E) The #endif for this directive is missing

C5038 (E)(W) Directive is not allowed -- an #else has already appeared

C5039 (E)(W) Division by zero

C5040 (E) Expected an identifier

C5041 (E) Expression must have arithmetic or pointer type

C5042 (E)(W) Operand types are incompatible ("type1" and "type2")

C5044 (E) Expression must have pointer type

C5045 (W) #undef may not be used on this predefined name

C5046 (W) "macro name" is predefined; attempted redefinition ignored

C5047 (W) Incompatible redefinition of macro "name" (declared at line "line number")

C5049 (E) Duplicate macro parameter name

C5050 (E) "##" may not be first in a macro definition

C5051 (E) "##" may not be last in a macro definition

C5052 (E) Expected a macro parameter name

C5053 (E) Expected a ":"

C5054 (W) Too few arguments in macro invocation

C5055 (W) Too many arguments in macro invocation

C5056 (E) Operand of sizeof may not be a function

C5057 (E) This operator is not allowed in a constant expression

C5058 (E) This operator is not allowed in a preprocessing expression

C5059 (E) Function call is not allowed in a constant expression

C5060 (E) This operator is not allowed in an integral constant expression

C5061 (W) Integer operation result is out of range

C5062 (W) Shift count is negative

C5063 (W) Shift count is too large

C5064 (W) Declaration does not declare anything

C5065 (E) Expected a ";"

C5066 (E) Enumeration value is out of "int" range

C5067 (E) Expected a "}"

C5068 (W) Integer conversion resulted in a change of sign

C5069 (W) Integer conversion resulted in truncation

C5070 (E) Incomplete type is not allowed

C5071 (E) Operand of sizeof may not be a bit field

C5075 (E) Operand of "*" must be a pointer

C5076 (W) Argument to macro is empty

C5077 (E) This declaration has no storage class or type specifier

C5078 (E) A parameter declaration may not have an initializer

C5079 (E) Expected a type specifier

C5080 (E) (W) A storage class may not be specified here

C5081 (E) More than one storage class may not be specified

C5082 (W) Storage class is not first

C5083 (W) Type qualifier specified more than once

C5084 (E) Invalid combination of type specifiers

C5085 (W) Invalid storage class for a parameter

C5086 (E) Invalid storage class for a function

C5087 (E) A type specifier may not be used here

C5088 (E) Array of functions is not allowed

C5089 (E) Array of void is not allowed

C5090 (E) Function returning function is not allowed

C5091 (E) Function returning array is not allowed

C5092 (E) Identifier-list parameters may only be used in a function definition

C5093 (E) Function type may not come from a typedef

C5094 (E) The size of an array must be greater than zero

C5095 (E) Array is too large

C5096 (W) A translation unit must contain at least one declaration

C5097 (E) A function may not return a value of this type

C5098 (E) An array may not have elements of this type

C5099 (E)(W) A declaration here must declare a parameter

C5100 (E) Duplicate parameter name

C5101 (E) "name" has already been declared in the current scope

C5102 (E) Forward declaration of enum type is nonstandard

C5103 (E) Class is too large

C5104 (E) Struct or union is too large

C5105 (E) Invalid size for bit field

C5106 (E) Invalid type for a bit field

C5107 (E)(W) Zero-length bit field must be unnamed

C5108 (W) Signed bit field of length 1

C5109 (E) Expression must have (pointer-to-) function type

C5110 (E) Expected either a definition or a tag name

C5111 (W) Statement is unreachable

C5112 (E) Expected "while"

C5114 (E)(W) Entity-kind "name" was referenced but not defined

C5115 (E) A continue statement may only be used within a loop

C5116 (E) A break statement may only be used within a loop or switch

C5117 (W) Non-void entity-kind "name" should return a value

C5118 (E) A void function may not return a value

C5119 (E) Cast to type "type" is not allowed

C5120 (E) Return value type does not match the function type

C5121 (E) A case label may only be used within a switch

C5122 (E) A default label may only be used within a switch

C5123 (E) Case label value has already appeared in this switch

C5124 (E) Default label has already appeared in this switch

C5125 (E) Expected a "("

C5126 (E) Expression must be an lvalue

C5127 (E) Expected a statement

C5128 (W) Loop is not reachable from preceding code

C5129 (E) A block-scope function may only have extern storage class

C5130 (E) Expected a "{"

C5131 (E) Expression must have pointer-to-class type

C5132 (E) Expression must have pointer-to-struct-or-union type

C5133 (E) Expected a member name

C5134 (E) Expected a field name

C5135 (E) Entity-kind "name" has no member "member name"

C5136 (E) Entity-kind "name" has no field "field name"

C5137 (E)(W) Expression must be a modifiable lvalue

C5138 (E)(W) Taking the address of a register field is not allowed

C5139 (E) Taking the address of a bit field is not allowed

C5140 (E)(W) Too many arguments in function call

C5141 (E) Unnamed prototyped parameters not allowed when body is present

C5142 (E) Expression must have pointer-to-object type

C5143 (F) Program too large or complicated to compile

C5144 (E) A value of type "type1" cannot be used to initialize an entity of type "type2"

C5145 (E) Entity-kind "name" may not be initialized

C5146 (E) Too many initializer values

C5147 (E)(W) Declaration is incompatible with "name" (declared at line "line number")

C5148 (E) Entity-kind "name" has already been initialized

C5149 (E) A global-scope declaration may not have this storage class

C5150 (E) A type name may not be redeclared as a parameter

C5151 (E) A typedef name may not be redeclared as a parameter

C5152 (W) Conversion of nonzero integer to pointer

C5153 (E) Expression must have class type

C5154 (E) Expression must have struct or union type

C5155 (W) Old-fashioned assignment operator

C5156 (W) Old-fashioned initializer

C5157 (E)(W) Expression must be an integral constant expression

C5158 (E) Expression must be an lvalue or a function designator

C5159 (E) Declaration is incompatible with previous "name" (declared at line "line number")

C5160 (E) Name conflicts with previously used external name "name"

C5161 (W) Unrecognized #pragma

C5163 (F) Could not open temporary file "name"

C5164 (F) Name of directory for temporary files is too long ("name")

C5165 (E) Too few arguments in function call

C5166 (E) Invalid floating constant

C5167 (E) Argument of type "type1" is incompatible with parameter of type "type2"

C5168 (E) A function type is not allowed here

C5169 (E) (W) Expected a declaration

C5170 (W) Pointer points outside of underlying object

C5171 (E) Invalid type conversion

C5172 (W)(I) External/internal linkage conflict with previous declaration

C5173 (E)(W) Floating-point value does not fit in required integral type

C5174 (I) Expression has no effect

C5175 (E)(W) Subscript out of range

C5177 (W) Entity-kind "name" was declared but never referenced

C5178 (W) "&" applied to an array has no effect

C5179 (W) Right operand of "%" is zero

C5180 (W)(I) Argument is incompatible with formal parameter

C5181 (W) Argument is incompatible with corresponding format string conversion

C5182 (F) Could not open source file "name" (no directories in search list)

C5183 (E) Type of cast must be integral

C5184 (E) Type of cast must be arithmetic or pointer

C5185 (I) Dynamic initialization in unreachable code

C5186 (W) Pointless comparison of unsigned integer with zero

C5187 (I) Use of "=" where "==" may have been intended

C5188 (W) Enumerated type mixed with another type

C5189 (F) Error while writing "file name" file

C5190 (F) Invalid intermediate language file

C5191 (W) Type qualifier is meaningless on cast type

C5192 (W) Unrecognized character escape sequence

C5193 (I) Zero used for undefined preprocessing identifier

C5194 (E) Expected an asm string

C5195 (E) An asm function must be prototyped

C5196 (E) An asm function may not have an ellipsis

C5219 (F) Error while deleting file "file name"

C5220 (E) Integral value does not fit in required floating-point type

C5221 (E) Floating-point value does not fit in required floating-point type

C5222 (E) Floating-point operation result is out of range

C5223 (W) Function function name declared implicitly

C5224 (W) The format string requires additional arguments

C5225 (W) The format string ends before this argument

C5226 (W) Invalid format string conversion

C5227 (E) Macro recursion

C5228 (W) Trailing comma is nonstandard

C5229 (W) Bit field cannot contain all values of the enumerated type

C5230 (W) Nonstandard type for a bit field

C5231 (W) Declaration is not visible outside of function

C5232 (W) Old-fashioned typedef of "void" ignored

C5233 (W) Left operand is not a struct or union containing this field

C5234 (W) Pointer does not point to struct or union containing this field

C5235 (E) Variable "name" was declared with a never-completed type

C5236 (W) (I) Controlling expression is constant

C5237 (I) Selector expression is constant

C5238 (E) Invalid specifier on a parameter

C5239 (E) Invalid specifier outside a class declaration

C5240 (E) Duplicate specifier in declaration

C5241 (E) A union is not allowed to have a base class

C5242 (E) Multiple access control specifiers are not allowed

C5243 (E) Class or struct definition is missing

C5244 (E) Qualified name is not a member of class "type" or its base classes

C5245 (E) A nonstatic member reference must be relative to a specific object

C5246 (E) A nonstatic data member may not be defined outside its class

C5247 (E) Entity-kind "name" has already been defined

C5248 (E) Pointer to reference is not allowed

C5249 (E) Reference to reference is not allowed

C5250 (E) Reference to void is not allowed

C5251 (E) Array of reference is not allowed

C5252 (E) Reference entity-kind "name" requires an initializer

C5253 (E) Expected a ","

C5254 (E) Type name is not allowed

C5255 (E) Type definition is not allowed

C5256 (E) Invalid redeclaration of type name "name" (declared at line "line number")

C5257 (E) Const entity-kind "name" requires an initializer

C5258 (E) "this" may only be used inside a nonstatic member function

C5259 (E) Constant value is not known

C5260 (W) Explicit type is missing ("int" assumed)

C5261 (I) Access control not specified ("name" by default)

C5262 (E)(W) Not a class or struct name

C5263 (E) Duplicate base class name

C5264 (E) Invalid base class

C5265 (E) Entity-kind "name" is inaccessible

C5266 (E) "name" is ambiguous

C5268 (E) Declaration may not appear after executable statement in block

C5269 (E) Conversion to inaccessible base class "type" is not allowed

C5274 (E) Improperly terminated macro invocation

C5276 (E) Name followed by "::" must be a class or namespace name

C5277 (E) Invalid friend declaration

C5278 (E) A constructor or destructor may not return a value

C5279 (E) Invalid destructor declaration

C5280 (E)(W) Declaration of a member with the same name as its class

C5281 (E) Global-scope qualifier (leading "::") is not allowed

C5282 (E) The global scope has no "name"

C5283 (E) Qualified name is not allowed

C5284 (E)(W) NULL reference is not allowed

C5285 (E) Initialization with "{...}" is not allowed for object of type "type"

C5286 (E) Base class "type" is ambiguous

C5287 (E) Derived class "type" contains more than one instance of class "type"

C5288 (E) Cannot convert pointer to base class "type1" to pointer to derived class "type2" – base class is virtual

C5289 (E) No instance of constructor "name" matches the argument list

C5290 (E) Copy constructor for class "type" is ambiguous

C5291 (E) No default constructor exists for class "type"

C5292 (E) "name" is not a nonstatic data member or base class of class "type"

C5293 (E) Indirect nonvirtual base class is not allowed

C5294 (E) Invalid union member -- class "type" has a disallowed member function

C5296 (E)(W) Invalid use of non-lvalue array

C5297 (E) Expected an operator

C5298 (E) Inherited member is not allowed

C5299 (E) Cannot determine which instance of entity-kind "name" is intended

C5300 (E)(W) A pointer to a bound function may only be used to call the function

C5301 (E) Typedef name has already been declared (with same type)

C5302 (E) Entity-kind "name" has already been defined

C5304 (E) No instance of entity-kind "name" matches the argument list

C5305 (E) Type definition is not allowed in function return type declaration

C5306 (E) Default argument not at end of parameter list

C5307 (E) Redefinition of default argument

C5308 (E) More than one instance of "name" matches the argument list:

C5309 (E) More than one instance of constructor "name" matches the argument list:

C5310 (E) Default argument of type "type1" is incompatible with parameter of type "type2"

C5311 (E) Cannot overload functions distinguished by return type alone

C5312 (E) No suitable user-defined conversion from "type1" to "type2" exists

C5313 (E) Type qualifier is not allowed on this function

C5314 (E) Only nonstatic member functions may be virtual

C5315 (E) The object has cv-qualifiers that are not compatible with the member function

C5316 (E) Program too large to compile (too many virtual functions)

C5317 (E) Return type is not identical to nor covariant with return type "type" of overridden virtual function entity-kind "name"

C5318 (E) Override of virtual entity-kind "name" is ambiguous

C5319 (E) Pure specifier ("= 0") allowed only on virtual functions

C5320 (E) Badly-formed pure specifier (only "= 0" is allowed)

C5321 (E) Data member initializer is not allowed

C5322 (E) Object of abstract class type "type" is not allowed:

C5323 (E) Function returning abstract class "type" is not allowed:

C5324 (I) Duplicate friend declaration

C5325 (E) Inline specifier allowed on function declarations only

C5326 (E)(W) "inline" is not allowed

C5327 (E) Invalid storage class for an inline function

C5328 (E) Invalid storage class for a class member

C5329 (E) Local class member entity-kind "name" requires a definition

C5330 (E) Entity-kind "name" is inaccessible

C5332 (E) Class "type" has no copy constructor to copy a const object

C5333 (E) Defining an implicitly declared member function is not allowed

C5334 (E) Class "type" has no suitable copy constructor

C5335 (E) (W) Linkage specification is not allowed

C5336 (E) Unknown external linkage specification

C5337 (E) Linkage specification is incompatible with previous "name" (declared at line "line number")

C5338 (E) More than one instance of overloaded function "name" has "C" linkage

C5339 (E) Class "type" has more than one default constructor

C5340 (E) Value copied to temporary, reference to temporary used

C5341 (E) "operator" must be a member function

C5342 (E) Operator may not be a static member function

C5343 (E) No arguments allowed on user-defined conversion

C5344 (E) Too many parameters for this operator function

C5345 (E) Too few parameters for this operator function

C5346 (E) Nonmember operator requires a parameter with class type

C5347 (E) Default argument is not allowed

C5348 (E) More than one user-defined conversion from "type1" to "type2" applies:

C5349 (E) No operator "operator" matches these operands

C5350 (E) More than one operator "operator" matches these operands:

C5351 (E) First parameter of allocation function must be of type "size_t"

C5352 (E) Allocation function requires "void *" return type

C5353 (E) Deallocation function requires "void" return type

C5354 (E) First parameter of deallocation function must be of type "void *"

C5356 (E) Type must be an object type

C5357 (E) Base class "type" has already been initialized

C5359 (E) Entity-kind "name" has already been initialized

C5360 (E) Name of member or base class is missing

C5363 (E) Invalid anonymous union -- nonpublic member is not allowed

C5364 (E) Invalid anonymous union -- member function is not allowed

C5365 (E) Anonymous union at global or namespace scope must be declared static

C5366 (E) Entity-kind "name" provides no initializer for:

C5367 (E) Implicitly generated constructor for class "type" cannot initialize:

C5368 (W) Entity-kind "name" defines no constructor to initialize the following:

C5369 (E) Entity-kind "name" has an uninitialized const or reference member

C5370 (W) Entity-kind "name" has an uninitialized const field

C5371 (E) Class "type" has no assignment operator to copy a const object

C5372 (E) Class "type" has no suitable assignment operator

C5373 (E) Ambiguous assignment operator for class "type"

C5375 (E) Declaration requires a typedef name

C5377 (W) "virtual" is not allowed

C5378 (E) "static" is not allowed

C5380 (E) Expression must have pointer-to-member type

C5381 (I) Extra ";" ignored

C5382 (W) In-class initializer for nonstatic member is nonstandard

C5384 (E) No instance of overloaded "name" matches the argument list

C5386 (E) No instance of entity-kind "name" matches the required type

C5388 (E) "operator->" for class "type1" returns invalid type "type2"

C5389 (E) A cast to abstract class "type" is not allowed:

C5390 (E) Function "main" may not be called or have its address taken

C5391 (E) A new-initializer may not be specified for an array

C5392 (E) Member function "name" may not be redeclared outside its class

C5393 (E) Pointer to incomplete class type is not allowed

C5394 (E) Reference to local variable of enclosing function is not allowed

C5397 (E) Implicitly generated assignment operator cannot copy:

C5398 (W) Cast to array type is nonstandard (treated as cast to "type")

C5399 (I) Entity-kind "name" has an operator newxxxx() but no default operator deletexxxx()

C5400 (I) Entity-kind "name" has a default operator deletexxxx() but no operator newxxxx()

C5401 (E) Destructor for base class "type" is not virtual

C5403 (E) Invalid redeclaration of member "function name"

C5404 (E) Function "main" may not be declared inline

C5405 (E) Member function with the same name as its class must be a constructor

C5407 (E) A destructor may not have parameters

C5408 (E) Copy constructor for class "type1" may not have a parameter of type "type2"

C5409 (E) Entity-kind "name" returns incomplete type "type"

C5410 (E) Protected entity-kind "name" is not accessible through a "type" pointer or object

C5411 (E) A parameter is not allowed

C5412 (E) An "asm" declaration is not allowed here

C5413 (E) No suitable conversion function from "type1" to "type2" exists

C5414 (W) Delete of pointer to incomplete class

C5415 (E) No suitable constructor exists to convert from "type1" to "type2"

C5416 (E) More than one constructor applies to convert from "type1" to "type2":

C5417 (E) More than one conversion function from "type1" to "type2" applies:

C5418 (E) More than one conversion function from "type" to a built-in type applies:

C5424 (E) A constructor or destructor may not have its address taken

C5427 (E) Qualified name is not allowed in member declaration

C5429 (E) The size of an array in "new" must be non-negative

C5430 (W) Returning reference to local temporary

C5432 (E) "enum" declaration is not allowed

C5433 (E) Qualifiers dropped in binding reference of type "type1" to initializer of type "type2"

C5434 (E) A reference of type "type1" (not const-qualified) cannot be initialized with a value of type "type2"

C5435 (E) A pointer to function may not be deleted

C5436 (E) Conversion function must be a nonstatic member function

C5437 (E) Template declaration is not allowed here

C5438 (E) Expected a "<"

C5439 (E) Expected a ">"

C5440 (E) Template parameter declaration is missing

C5441 (E) Argument list for entity-kind "name" is missing

C5442 (E) Too few arguments for entity-kind "name"

C5443 (E) Too many arguments for entity-kind "name"

C5445 (E) Entity-kind "name1" is not used in declaring the parameter types of entity-kind "name2"

C5449 (E) More than one instance of entity-kind "name" matches the required type

C5450 (E) The type "long long" is nonstandard

C5451 (E) Omission of "class" is nonstandard

C5452 (E) Return type may not be specified on a conversion function

C5456 (E) Excessive recursion at instantiation of entity-kind "name"

C5457 (E) "name" is not a function or static data member

C5458 (E) Argument of type "type1" is incompatible with template parameter of type "type2"

C5459 (E) Initialization requiring a temporary or conversion is not allowed

C5460 (W) Declaration of "variable name" hides function parameter

C5461 (E) Initial value of reference to non-const must be an lvalue

C5463 (E) "template" is not allowed

C5464 (E) "type" is not a class template

C5466 (E) "main" is not a valid name for a function template

C5467 (E) Invalid reference to entity-kind "name" (union/nonunion mismatch)

C5468 (E) A template argument may not reference a local type

C5469 (E) Tag kind of "name1" is incompatible with declaration of entity-kind "name2" (declared at line "line number")

C5470 (E) The global scope has no tag named "name"

C5471 (E) Entity-kind "name1" has no tag member named "name2"

C5473 (E) Entity-kind "name" may be used only in pointer-to-member declaration

C5475 (E) A template argument may not reference a non-external entity

C5476 (E) Name followed by "::~" must be a class name or a type name

C5477 (E) Destructor name does not match name of class "type"

C5478 (E) Type used as destructor name does not match type "type"

C5479 (I) Entity-kind "name" redeclared "inline" after being called

C5481 (E) Invalid storage class for a template declaration

C5484 (E) Invalid explicit instantiation declaration

C5485 (E) Entity-kind "name" is not an entity that can be instantiated

C5486 (E) Compiler generated entity-kind "name" cannot be explicitly instantiated

C5487 (E)(I) Inline entity-kind "name" cannot be explicitly instantiated

C5489 (E) Entity-kind "name" cannot be instantiated -- no template definition was supplied

C5490 (E) Entity-kind "name" cannot be instantiated -- it has been explicitly specialized

C5493 (E) No instance of entity-kind "name" matches the specified type

C5494 (E)(W) Declaring a void parameter list with a typedef is nonstandard

C5496 (E) Template parameter "name" may not be redeclared in this scope

C5497 (W) Declaration of "name" hides template parameter

C5498 (E) Template argument list must match the parameter list

C5500 (E) Extra parameter of postfix "operatorxxxx" must be of type "int"

C5501 (E) An operator name must be declared as a function

C5502 (E) Operator name is not allowed

C5503 (E) Entity-kind "name" cannot be specialized in the current scope

C5504 (E) Nonstandard form for taking the address of a member function

C5505 (E) Too few template parameters -- does not match previous declaration

C5506 (E) Too many template parameters -- does not match previous declaration

C5507 (E) Function template for operator delete(void *) is not allowed

C5508 (E) Class template and template parameter may not have the same name

C5510 (E) A template argument may not reference an unnamed type

C5511 (E) Enumerated type is not allowed

C5512 (W) Type qualifier on a reference type is not allowed

C5513 (E)(W) A value of type "type1" cannot be assigned to an entity of type "type2"

C5514 (W) Pointless comparison of unsigned integer with a negative constant

C5515 (E) Cannot convert to incomplete class "type"

C5516 (E) Const object requires an initializer

C5517 (E) Object has an uninitialized const or reference member

C5518 (E) Nonstandard preprocessing directive

C5519 (E) Entity-kind "name" may not have a template argument list

C5520 (E)(W) Initialization with "{...}" expected for aggregate object

C5521 (E) Pointer-to-member selection class types are incompatible ("type1" and "type2")

C5522 (W) Pointless friend declaration

C5523 (W) "." used in place of "::" to form a qualified name

C5525 (W) A dependent statement may not be a declaration

C5526 (E) A parameter may not have void type

C5529 (E) This operator is not allowed in a template argument expression

C5530 (E) Try block requires at least one handler

C5531 (E) Handler requires an exception declaration

C5532 (E) Handler is masked by default handler

C5533 (W) Handler is potentially masked by previous handler for type "type"

C5534 (I) Use of a local type to specify an exception

C5535 (I) Redundant type in exception specification

C5536 (E) Exception specification is incompatible with that of previous entity-kind "name" (declared at line "line number"):

C5540 (E) Support for exception handling is disabled

C5541 (W) Omission of exception specification is incompatible with previous entity-kind "name" (declared at line "line number")

C5542 (F) Could not create instantiation request file "name"

C5543 (E) Non-arithmetic operation not allowed in nontype template argument

C5544 (E) Use of a local type to declare a nonlocal variable

C5545 (E) Use of a local type to declare a function

C5546 (E) Transfer of control bypasses initialization of:

C5548 (E) Transfer of control into an exception handler

C5549 (I) Entity-kind "name" is used before its value is set

C5550 (W) Entity-kind "name" was set but never used

C5551 (E) Entity-kind "name" cannot be defined in the current scope

C5552 (W) Exception specification is not allowed

C5553 (W) External/internal linkage conflict for entity-kind "name" (declared at line "line number")

C5554 (W) Entity-kind "name" will not be called for implicit or explicit conversions

C5555 (E) Tag kind of "name" is incompatible with template parameter of type "type"

C5556 (E) Function template for operator new(size_t) is not allowed

C5558 (E) Pointer to member of type "type" is not allowed

C5559 (E) Ellipsis is not allowed in operator function parameter list

C5560 (E) "keyword" is reserved for future use as a keyword

C5563 (F) Invalid preprocessor output file

C5598 (E) A template parameter may not have void type

C5599 (E) Excessive recursive instantiation of entity-kind "name" due to instantiate-all mode

C5601 (E) A throw expression may not have void type

C5603 (E) Parameter of abstract class type "type" is not allowed:

C5604 (E) Array of abstract class "type" is not allowed:

C5605 (E) Floating-point template parameter is nonstandard

C5606 (E) This pragma must immediately precede a declaration

C5607 (E) This pragma must immediately precede a statement

C5608 (E) This pragma must immediately precede a declaration or statement

C5609 (E) This kind of pragma may not be used here

C5611 (W) Overloaded virtual function "name1" is only partially overridden in entity-kind "name2"

C5612 (E) Specific definition of inline template function must precede its first use

C5615 (E) Parameter type involves pointer to array of unknown bound

C5616 (E) Parameter type involves reference to array of unknown bound

C5617 (W) Pointer-to-member-function cast to pointer to function

C5618 (I) Struct or union declares no named members

C5619 (E) Nonstandard unnamed field

C5620 (E) Nonstandard unnamed member

C5624 (E) "name" is not a type name

C5641 (F) "name" is not a valid directory

C5642 (F) Cannot build temporary file name

C5643 (E) "restrict" is not allowed

C5644 (E) A pointer or reference to function type may not be qualified by "restrict"

C5647 (E) Conflicting calling convention modifiers

C5650 (W) Calling convention specified here is ignored

C5651 (E) A calling convention may not be followed by a nested declarator

C5652 (I) Calling convention is ignored for this type

C5654 (E) Declaration modifiers are incompatible with previous declaration

C5656 (E) Transfer of control into a try block

C5657 (W) Inline specification is incompatible with previous "name" (declared at line "line number")

C5658 (E) Closing brace of template definition not found

C5660 (E) Invalid packing alignment value

C5661 (E) Expected an integer constant

C5662 (W) Call of pure virtual function

C5663 (E) Invalid source file identifier string

C5664 (E) A class template cannot be defined in a friend declaration

C5665 (E) "asm" is not allowed

C5666 (E) "asm" must be used with a function definition

C5667 (E) "asm" function is nonstandard

C5668 (E) Ellipsis with no explicit parameters is nonstandard

C5669 (E) "&..." is nonstandard

C5670 (E) Invalid use of "&..."

C5673 (E) A reference of type "type1" cannot be initialized with a value of type "type2"

C5674 (E) Initial value of reference to const volatile must be an lvalue

C5676 (W) Using out-of-scope declaration of "symbol name"

C5678 (I) Call of entity-kind "name" (declared at line "line number") cannot be inlined

C5679 (I) Entity-kind "name" cannot be inlined

C5691 (E)(W) "symbol", required for copy that was eliminated, is inaccessible

C5692 (E)(W) "symbol", required for copy that was eliminated, is not callable because reference parameter cannot be bound to rvalue

C5693 (E) <typeinfo> must be included before typeid is used

C5694 (E) "name" cannot cast away const or other type qualifiers

C5695 (E) The type in a dynamic_cast must be a pointer or reference to a complete class type, or void *

C5696 (E) The operand of a pointer dynamic_cast must be a pointer to a complete class type

C5697 (E) The operand of a reference dynamic_cast must be an lvalue of a complete class type

C5698 (E) The operand of a runtime dynamic_cast must have a polymorphic class type

C5701 (E) An array type is not allowed here

C5702 (E) Expected an "="

C5703 (E) Expected a declarator in condition declaration

C5704 (E) "name", declared in condition, may not be redeclared in this scope

C5705 (E) Default template arguments are not allowed for function templates

C5706 (E) Expected a "," or ">"

C5707 (E) Expected a template parameter list

C5708 (W) Incrementing a bool value is deprecated

C5709 (E) bool type is not allowed

C5710 (E) Offset of base class "name1" within class "name2" is too large

C5711 (E) Expression must have bool type (or be convertible to bool)

C5717 (E) The type in a const_cast must be a pointer, reference, or pointer to member to an object type

C5718 (E) A const_cast can only adjust type qualifiers; it cannot change the underlying type

C5719 (E) mutable is not allowed

C5720 (W) Redeclaration of entity-kind "name" is not allowed to alter its access

C5722 (W) Use of alternative token "<:" appears to be unintended

C5723 (W) Use of alternative token "%:" appears to be unintended

C5724 (E) namespace definition is not allowed

C5725 (E) Name must be a namespace name

C5726 (E) Namespace alias definition is not allowed

C5727 (E) namespace-qualified name is required

C5728 (E) A namespace name is not allowed

C5730 (E) Entity-kind "name" is not a class template

C5731 (E) Array with incomplete element type is nonstandard

C5732 (E) Allocation operator may not be declared in a namespace

C5733 (E) Deallocation operator may not be declared in a namespace

C5734 (E) Entity-kind "name1" conflicts with using-declaration of entity-kind "name2"

C5735 (E) Using-declaration of entity-kind "name1" conflicts with entity-kind "name2" (declared at line "line number")

C5737 (W) Using-declaration ignored -- it refers to the current namespace

C5738 (E) A class-qualified name is required

C5741 (W) Using-declaration of entity-kind "name" ignored

C5742 (E) Entity-kind "name1" has no actual member "name2"

C5748 (W) Calling convention specified more than once

C5749 (E) A type qualifier is not allowed

C5750 (E) Entity-kind "name" (declared at line "line number") was used before its template was declared

C5751 (E) Static and nonstatic member functions with same parameter types cannot be overloaded

C5752 (E) No prior declaration of entity-kind "name"

C5753 (E) A template-id is not allowed

C5754 (E) A class-qualified name is not allowed

C5755 (E) Entity-kind "name" may not be redeclared in the current scope

C5756 (E) Qualified name is not allowed in namespace member declaration

C5757 (E) Entity-kind "name" is not a type name

C5758 (E) Explicit instantiation is not allowed in the current scope

C5759 (E) "symbol name" cannot be explicitly instantiated in the current scope

C5760 (W) "symbol" explicitly instantiated more than once

C5761 (E) Typename may only be used within a template

C5765 (E) Nonstandard character at start of object-like macro definition

C5766 (W) Exception specification for virtual entity-kind "name1" is incompatible with that of overridden entity-kind "name2"

C5767 (W) Conversion from pointer to smaller integer

C5768 (W) Exception specification for implicitly declared virtual entity-kind "name1" is incompatible with that of overridden entity-kind "name2"

C5769 (E) "symbol1", implicitly called from "symbol2", is ambiguous

C5771 (E) "explicit" is not allowed

C5772 (E) Declaration conflicts with "name" (reserved class name)

C5773 (E) Only "()" is allowed as initializer for array entity-kind "name"

C5774 (E) "virtual" is not allowed in a function template declaration

C5775 (E) Invalid anonymous union -- class member template is not allowed

C5776 (E) Template nesting depth does not match the previous declaration of entity-kind "name"

C5777 (E) This declaration cannot have multiple "template <...>" clauses

C5779 (E) "name", declared in for-loop initialization, may not be redeclared in this scope

C5780 (W) Reference is to "symbol1" -- under old for-init scoping rules it would have been "symbol2"

C5782 (E) Definition of virtual entity-kind "name" is required here

C5783 (W) Empty comment interpreted as token-pasting operator "##"

C5784 (E) A storage class is not allowed in a friend declaration

C5785 (E) Template parameter list for "name" is not allowed in this declaration

C5786 (E) entity-kind "name" is not a valid member class or function template

C5787 (E) Not a valid member class or function template declaration

C5788 (E) A template declaration containing a template parameter list may not be followed by an explicit specialization declaration

C5789 (E) Explicit specialization of entity-kind "name1" must precede the first use of entity-kind "name2"

C5790 (E) Explicit specialization is not allowed in the current scope

C5791 (E) Partial specialization of entity-kind "name" is not allowed

C5792 (E) Entity-kind "name" is not an entity that can be explicitly specialized

C5793 (E) Explicit specialization of entity-kind "name" must precede its first use

C5794 (W) Template parameter "template parameter" may not be used in an elaborated type specifier

C5795 (E) Specializing "name" requires "template<>" syntax

C5799 (E) Specializing "symbol name" without "template<>" syntax is nonstandard

C5800 (E) This declaration may not have extern "C" linkage

C5801 (E) "name" is not a class or function template name in the current scope

C5802 (W) Specifying a default argument when redeclaring an unreferenced function template is nonstandard

C5803 (E) Specifying a default argument when redeclaring an already referenced function template is not allowed

C5804 (E) Cannot convert pointer to member of base class "type1" to pointer to member of derived class "type2" -- base class is virtual

C5805 (E) Exception specification is incompatible with that of entity-kind "name" (declared at line "line number"):

C5806 (W) Omission of exception specification is incompatible with entity-kind "name" (declared at line "line number")

C5807 (E) Unexpected end of default argument expression

C5808 (E) Default-initialization of reference is not allowed

C5809 (E) Uninitialized entity-kind "name" has a const member

C5810 (E) Uninitialized base class "type" has a const member

C5811 (E) Const entity-kind "name" requires an initializer -- class "type" has no explicitly declared default constructor

C5812 (E)(W) Const object requires an initializer -- class "type" has no explicitly declared default constructor

C5815 (I) Type qualifier on return type is meaningless

C5816 (E) In a function definition a type qualifier on a "void" return type is not allowed

C5817 (E) Static data member declaration is not allowed in this class

C5818 (E) Template instantiation resulted in an invalid function declaration

C5819 (E) "..." is not allowed

C5822 (E) Invalid destructor name for type "type"

C5824 (E) Destructor reference is ambiguous -- both entity-kind "name1" and entity-kind "name2" could be used

C5825 (W) Virtual inline entity-kind "name" was never defined

C5826 (W) Entity-kind "name" was never referenced

C5827 (E) Only one member of a union may be specified in a constructor initializer list

C5828 (E) Support for "new[]" and "delete[]" is disabled

C5829 (W) "double" used for "long double" in generated C code

C5830 (W) "symbol" has no corresponding operator deletes (to be called if an exception is thrown during initialization of an allocated object)

C5831 (W)(I) Support for placement delete is disabled

C5832 (E) No appropriate operator delete is visible

C5833 (E) Pointer or reference to incomplete type is not allowed

C5834 (E) Invalid partial specialization -- entity-kind "name" is already fully specialized

C5835 (E) Incompatible exception specifications

C5836 (W) Returning reference to local variable

C5837 (W) Omission of explicit type is nonstandard ("int" assumed)

C5838 (E) More than one partial specialization matches the template argument list of entity-kind "name"

C5840 (E) A template argument list is not allowed in a declaration of a primary template

C5841 (E) Partial specializations may not have default template arguments

C5842 (E) Entity-kind "name1" is not used in template argument list of entity-kind "name2"

C5843 (E) The type of partial specialization template parameter entity-kind "name" depends on another template parameter

C5844 (E) The template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter

C5845 (E) This partial specialization would have been used to instantiate entity-kind "name"

C5846 (E) This partial specialization would have been made the instantiation of entity-kind "name" ambiguous

C5847 (E) Expression must have integral or enum type

C5848 (E) Expression must have arithmetic or enum type

C5849 (E) Expression must have arithmetic, enum, or pointer type

C5850 (E) Type of cast must be integral or enum

C5851 (E) Type of cast must be arithmetic, enum, or pointer

C5852 (E) Expression must be a pointer to a complete object type

C5854 (E) A partial specialization nontype argument must be the name of a nontype parameter or a constant

C5855 (E)(W) Return type is not identical to return type "type" of overridden virtual function entity-kind "name"

C5857 (E) A partial specialization of a class template must be declared in the namespace of which it is a member

C5858 (E) Entity-kind "name" is a pure virtual function

C5859 (E) Pure virtual entity-kind "name" has no overrider

C5861 (E) Invalid character in input line

C5862 (E) Function returns incomplete type "type"

C5863 (I) Effect of this "#pragma pack" directive is local to "symbol"

C5864 (E) "name" is not a template

C5865 (E) A friend declaration may not declare a partial specialization

C5866 (I) Exception specification ignored

C5867 (W) Declaration of "size_t" does not match the expected type "type"

C5868 (E) Space required between adjacent ">" delimiters of nested template argument lists (">>" is the right shift operator)

C5869 (E) Could not set locale to allow processing of multibyte characters

C5870 (W) Invalid multibyte character sequence

C5871 (E) Template instantiation resulted in unexpected function type of "type1" (the meaning of a name may have changed since the template declaration -- the type of the template is "type2")

C5872 (E) Ambiguous guiding declaration -- more than one function template no matches type "type"

C5873 (E) Non-integral operation not allowed in nontype template argument

C5875 (E) Embedded C++ does not support templates

C5876 (E) Embedded C++ does not support exception handling

C5877 (E) Embedded C++ does not support namespaces

C5878 (E) Embedded C++ does not support run-time type information

C5879 (E) Embedded C++ does not support the new cast syntax

C5880 (E) Embedded C++ does not support using-declarations

C5881 (E) Embedded C++ does not support "mutable"

C5882 (E) Embedded C++ does not support multiple or virtual inheritance

C5885 (E) "type1" cannot be used to designate constructor for "type2"

C5886 (E) Invalid suffix on integral constant

C5890 (E) Variable length array with unspecified bound is not allowed

C5891 (E) An explicit template argument list is not allowed on this declaration

C5892 (E) An entity with linkage cannot have a type involving a variable length array

C5893 (E) A variable length array cannot have static storage duration

C5894 (E) Entity-kind "name" is not a template

C5896 (E) Expected a template argument

C5898 (E) Nonmember operator requires a parameter with class or enum type

C5900 (E) Using-declaration of entity-kind "name" is not allowed

C5901 (E) Qualifier of destructor name "type1" does not match type "type2"

C5902 (W) Type qualifier ignored

C5907 (E) Option "nonstd_qualifier_deduction" can be used only when compiling C++

C5912(W) Ambiguous class member reference - "symbol1" used in preference to "symbol2"

C5915 (E) A segment name has already been specified

C5916 (E) Cannot convert pointer to member of derived class "type1" to pointer to member of base class "type2" -- base class is virtual

C5919 (F) Invalid output file: "name"

C5920 (F) Cannot open output file: "name"

C5925 (W) Type qualifiers on function types are ignored

C5926 (F) Cannot open definition list file: "name"

C5928 (E) Incorrect use of va_start

C5929 (E) Incorrect use of va_arg

C5930 (E) Incorrect use of va_end

C5934 (E) A member with reference type is not allowed in a union

C5935 (E) "typedef" may not be specified here

C5936 (W) Redeclaration of entity-kind "name" alters its access

C5937 (E) A class or namespace qualified name is required

C5938 (E) Return type "int" omitted in declaration of function "main"

C5939 (E) pointer-to-member representation "symbol1" is too restrictive for "symbol2"

C5940 (W) Missing return statement at end of non-void entity-kind "name"

C5941 (W) Duplicate using-declaration of "name" ignored

C5942 (W) enum bit-fields are always unsigned, but enum "name" includes negative enumerator

C5946 (E) Name following "template" must be a member template

C5947 (E) Name following "template" must have a template argument list

C5948 (E)(W) Nonstandard local-class friend declaration -- no prior declaration in the enclosing scope

C5949 (I) Specifying a default argument on this declaration is nonstandard

C5951 (E)(W) Return type of function "main" must be "int"

C5952 (E) A template parameter may not have class type

C5953 (E) A default template argument cannot be specified on the declaration of a member of a class template

C5954 (E) A return statement is not allowed in a handler of a function try block of a constructor

C5955 (E) Ordinary and extended designators cannot be combined in an initializer designation

C5956 (E) The second subscript must not be smaller than the first

C5959 (W) Declared size for bit field is larger than the size of the bit field type; truncated to "number of bits" bits

C5960 (E) Type used as constructor name does not match type "type"

C5961 (W) Use of a type with no linkage to declare a variable with linkage

C5962 (W) Use of a type with no linkage to declare a function

C5963 (E) Return type may not be specified on a constructor

C5964 (E) Return type may not be specified on a destructor

C5965 (E) Incorrectly formed universal character name

C5966 (E) Universal character name specifies an invalid character

C5967 (E) A universal character name cannot designate a character in the basic character set

C5968 (E) This universal character is not allowed in an identifier

C5969 (E) The identifier __VA_ARGS__ can only appear in the replacement lists of variadic macros

C5970 (W) The qualifier on this friend declaration is ignored

C5971 (E) Array range designators cannot be applied to dynamic initializers

C5972 (E) Property name cannot appear here

C5973 (W) "inline" used as a function qualifier is ignored

C5975 (E) A variable-length array type is not allowed

C5976 (E) A compound literal is not allowed in an integral constant expression

C5977 (E) A compound literal of type "type" is not allowed

C5978 (E) A template friend declaration cannot be declared in a local class

C5979 (E) Ambiguous "?" operation: second operand of type "type1" can be converted to third operand type "type2", and vice versa

C5980 (E) Call of an object of a class type without appropriate operator() or conversion functions to pointer-to-function type

C5982 (E) There is more than one way an object of type "type" can be called for the argument list

C5983 (E) typedef name has already been declared (with similar type)

C5984 (W) Operator new and operator delete cannot be given internal linkage

C5985 (E) Storage class "mutable" is not allowed for anonymous unions

C5987 (E) Abstract class type "type" is not allowed as catch type:

C5988 (E) A qualified function type cannot be used to declare a nonmember function or a static member function

C5989 (E) A qualified function type cannot be used to declare a parameter

C5990 (E) Cannot create a pointer or reference to qualified function type

C5991 (W) Extra braces are nonstandard

C5992 (E) Invalid macro definition:

C5993 (W) Subtraction of pointer types "symbol name1" and "symbol name2" is nonstandard

C5994 (E) An empty template parameter list is not allowed in a template parameter declaration

C5995 (E) Expected "class"

C5996 (E) The "class" keyword must be used when declaring a template parameter

C5997 (W) "function name1" is hidden by "function name2" -- virtual function override intended?

C5998 (E) A qualified name is not allowed for a friend declaration that is a function definition

C5999 (E) "type1" is not compatible with "type2"

C6000 (W) A storage class may not be specified here

C6001 (E) Class member designated by a using-declaration must be visible in a direct base class

C6006 (E) A template parameter cannot have the same name as one of its template parameters

C6007 (E) Recursive instantiation of default argument

C6009 (E) "instance name" is not an entity that can be defined

C6010 (E) Destructor name must be qualified

C6011 (E) Friend class name may not be introduced with "typename"

C6012 (E) A using-declaration may not name a constructor or destructor

C6013 (E) A qualified friend template declaration must refer to a specific previously declared template

C6014 (E) Invalid specifier in class template declaration

C6015 (E) Argument is incompatible with formal parameter

C6017 (E) Loop in sequence of "operator->" functions starting at class "symbol"

C6018 (E) "class name" has no member class "member name"

C6019 (E) The global scope has no class named "class name"

C6020 (E) Recursive instantiation of template default argument

C6021 (E) Access declarations and using-declarations cannot appear in unions

C6022 (E) "name" is not a class member

C6023 (E) Nonstandard member constant declaration is not allowed

C6028 (W) Invalid redeclaration of nested class

C6029 (E) Type containing an unknown-size array is not allowed

C6030 (W) A variable with static storage duration cannot be defined within an inline function

C6031 (W) An entity with internal linkage cannot be referenced within an inline function with external linkage

C6032 (E) Argument type "type" does not match this type-generic function macro

C6034 (E) Friend declaration cannot add default arguments to previous declaration

C6035 (E) "template name" cannot be declared in this scope

C6036 (E) The reserved identifier "symbol" may only be used inside a function

C6037 (E) This universal character cannot begin an identifier

C6038 (E) Expected a string literal

C6039 (E) Unrecognized STDC pragma

C6040 (E) Expected "ON", "OFF", or "DEFAULT"

C6041 (E) A STDC pragma may only appear between declarations in the global scope or before any statements or declarations in a block scope

C6042 (E) Incorrect use of va_copy

C6043 (E) "type" can only be used with floating-point types

C6044 (E) Complex type is not allowed

C6045 (E) Invalid designator kind

C6046 (W) Floating-point value cannot be represented exactly

C6047 (E) Complex floating-point operation result is out of range

C6048 (E) Conversion between real and imaginary yields zero

C6049 (E) An initializer cannot be specified for a flexible array member

C6050 (W) imaginary *= imaginary sets the left-hand operand to zero

C6051 (E)(W) Standard requires that "symbol" be given a type by a subsequent declaration ("int" assumed)

C6052 (E) A definition is required for inline "symbol"

C6053 (W) Conversion from integer to smaller pointer

C6054 (E) A floating-point type must be included in the type specifier for a _Complex or _Imaginary type

C6055 (E) Types cannot be declared in anonymous unions

C6056 (W) Returning pointer to local variable

C6057 (W) Returning pointer to local temporary

C6061 (E) Declaration of "symbol name" is incompatible with a declaration in another translation unit

C6062 (E) The other declaration is "line"

C6065 (E) A field declaration cannot have a type involving a variable length array

C6066 (E) declaration of "instance" had a different meaning during compilation of "symbol"

C6067 (E) Expected "template"

C6072 (E)(W) A declaration cannot have a label

C6075 (E) "instance name" already defined during compilation of "symbol"

C6076 (E) "symbol" already defined in another translation unit

C6081 (E) A field with the same name as its class cannot be declared in a class with a user-declared constructor

C6083 (F) Exported template file file name is corrupted

C6086 (E) the object has cv-qualifiers that are not compatible with the member "symbol"

C6087 (E) No instance of "class name" matches the argument list and object (the object has cv-qualifiers that prevent a match)

C6089 (E) There is no type with the width specified

C6105 (W) #warning directive: "character/string"

C6139 (E) The "template" keyword used for syntactic disambiguation may only be used within a template

C6144 (E) Storage class must be auto or register

C6145 (W) "type1" would have been promoted to "type2" when passed through the ellipsis parameter; use the latter type instead

C6146 (E) "symbol" is not a base class member

C6151 (F) Mangled name is too long

C6158 (E) void return type cannot be qualified

C6161 (E) A member template corresponding to "symbol" is declared as a template of a different kind in another translation unit

C6163 (E) va_start should only appear in a function with an ellipsis parameter

C6192 (W) Null (zero) character in input line ignored

C6193 (W) Null (zero) character in string or character constant

C6194 (W) Null (zero) character in header name

C6197 (W) The prototype declaration of "symbol" is ignored after this unprototyped redeclaration

C6201 (E) Typedef "symbol" may not be used in an elaborated type specifier

C6203 (E) Parameter "parameter name" may not be redeclared in a catch clause of function try block

C6204 (E) The initial explicit specialization of "symbol name" must be declared in the namespace containing the template

C6206 (E) "template" must be followed by an identifier

C6211 (W) Nonstandard cast to array type ignored

C6212 (E) This pragma cannot be used in a _Pragma operator (a #pragma directive must be used)

C6213 (W) Field uses tail padding of a base class

C6218 (W) Base class "class name1" uses tail padding of base class "class name2"

C6222 (W) Invalid error number

C6223 (W) Invalid error tag

C6224 (W) Expected an error number or error tag

C6227 (E) Transfer of control into a statement expression is not allowed

C6229 (E) This statement is not allowed inside of a statement expression

C6230 (E) A non-POD class definition is not allowed inside of a statement expression

C6235 (W) Nonstandard conversion between pointer to function and pointer to data

C6254 (E) Integer overflow in internal computation due to size or complexity of "type"

C6255 (E) Integer overflow in internal computation

C6273 (W) Alignment-of operator applied to incomplete type

C6280 (E) Conversion from inaccessible base class "class name" is not allowed

C6282 (E) String literals with different character kinds cannot be concatenated

C6285 (W) Nonstandard qualified name in namespace member declaration

C6290 (W) Non-POD class type passed through ellipsis

C6291 (E) A non-POD class type cannot be fetched by va_arg

C6292 (E) The 'u' or 'U' suffix must appear before the 'l' or 'L' suffix in a fixed-point literal

C6294 (W) Integer operand may cause fixed-point overflow

C6295 (E) Fixed-point constant is out of range

C6296 (W) Fixed-point value cannot be represented exactly

C6297 (W) Constant is too large for long long; given unsigned long long type (nonstandard)

C6301 (W) "symbol" declares a non-template function -- add <> to refer to a template instance

C6302 (W) Operation may cause fixed-point overflow

C6303 (E) Expression must have integral, enum, or fixed-point type

C6304 (E) Expression must have integral or fixed-point type

C6307 (W) Class member typedef may not be redeclared

C6308 (W) Taking the address of a temporary

C6310 (W) Fixed-point value implicitly converted to floating-point type

C6311 (E) Fixed-point types have no classification

C6312 (E) A template parameter may not have fixed-point type

C6313 (E) Hexadecimal floating-point constants are not allowed

C6315 (E) Floating-point value does not fit in required fixed-point type

C6316 (W) Value cannot be converted to fixed-point value exactly

C6317 (E) Fixed-point conversion resulted in a change of sign

C6318 (E) Integer value does not fit in required fixed-point type

C6319 (E)(W) Fixed-point operation result is out of range

C6320 (E) Multiple named address spaces

C6321 (E) Variable with automatic storage duration cannot be stored in a named address space

C6322 (E) Type cannot be qualified with named address space

C6323 (E) Function type cannot be qualified with named address space

C6324 (E) Field type cannot be qualified with named address space

C6325 (E) Fixed-point value does not fit in required floating-point type

C6326 (E) Fixed-point value does not fit in required integer type

C6327 (E) Value does not fit in required fixed-point type

C6335 (F) Cannot open predefined macro file: "file name"

C6336 (F) Invalid predefined macro entry at line "line count": "macro name"

C6337 (F) Invalid macro mode name "macro mode name"

C6338 (F) Incompatible redefinition of predefined macro "macro name"

C6342 (W) const_cast to enum type is nonstandard

C6344 (E) A named address space qualifier is not allowed here

C6345 (E) An empty initializer is invalid for an array with unspecified bound

C6346 (W) Function returns incomplete class type "class name"

C6348 (I) Declaration hides "variable name"

C6349 (E) A parameter cannot be allocated in a named address space

C6350 (E) Invalid suffix on fixed-point or floating-point constant

C6351 (E) A register variable cannot be allocated in a named address space

C6352 (E) Expected "SAT" or "DEFAULT"

C6353 (I) "symbol name" has no corresponding member operator delete "symbol name" (to be called if an exception is thrown during initialization of an allocated object)

C6355 (E) A function return type cannot be qualified with a named address space

C6361 (W) Negation of an unsigned fixed-point value

C6365 (E) Named-register variables cannot have void type

C6372 (E) Nonstandard qualified name in global scope declaration

C6373 (W) Implicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)

C6374 (W) Explicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)

C6375 (W) Conversion from pointer to same-sized integral type (potential portability problem)

C6380 (E)(I) Virtual "function name" was not defined (and cannot be defined elsewhere because it is a member of an unnamed namespace)

C6381 (E)(I) Carriage return character in source line outside of comment or character/string literal

C6382 (E) Expression must have fixed-point type

C6386 (W) Storage specifier ignored

C6396 (W) White space between backslash and newline in line splice ignored

C6398 (E) Invalid member for anonymous member class -- class "symbol" has a disallowed member function

C6400 (W) Positional format specifier cannot be zero

C6403 (E) A variable-length array is not allowed in a function return type

C6404 (E) Variable-length array type is not allowed in pointer to member of type "type"

C6405 (E) The result of a statement expression cannot have a type involving a variable-length array

C6420 (E)(W) Some enumerator values cannot be represented by the integral type underlying the enum type

C6421 (E) Default argument is not allowed on a friend class template declaration

C6422 (W) Multicharacter character literal (potential portability problem)

C6424 (E) Second operand of offsetof must be a field

C6425 (E) Second operand of offsetof may not be a bit field

C6426 (E) Cannot apply offsetof to a member of a virtual base

C6427 (W) offsetof applied to non-POD types is nonstandard

C6428 (E) Default arguments are not allowed on a friend declaration of a member function

C6429 (E) Default arguments are not allowed on friend declarations that are not definitions

C6430 (E) Redeclaration of "function name" previously declared as a friend with default arguments is not allowed

C6431 (E) Invalid qualifier for "symbol" (a derived class is not allowed here)

C6432 (E) Invalid qualifier for definition of class "class name"

C6439 (E) Template argument list of "symbol" must match the parameter list

C6440 (E) An incomplete class type is not allowed

C6445 (E) Invalid redefinition of "symbol name"

C6449 (E) Explicit specialization of "symbol" must precede its first use "symbol2"

C6623 (W) The destructor for "class1" has been suppressed because the destructor for "class2" is inaccessible

C6648 (W) '=' assumed following macro name "macro name" in command-line definition

C6649 (E)(W) White space is required between the macro name "macro name" and its replacement text

C6655 (E) "symbol" cannot be declared inline after its definition "definition name"

C6671 (W) __assume expression with side effects discarded

C6674 (E) __evenaccess qualifier is applied to only integer type

C6675 (E) Expected a section name string

C6676 (E) Expected a section name

C6677 (E) Invalid pragma declaration

C6678 (E) "symbol name" has already been specified by other pragma

C6679 (E) Pragma may not be specified after definition

C6680 (E) Invalid kind of pragma is specified to this symbol

C6681 (I) This pragma has no effect

C6682 (E) "symbol name" must be qualified for function type

C6683 (E) Illegal "pragma name" specifier

C6684 (E) Multiple pointer qualifiers

C6685 (E) __ptr16 must be qualified for data pointer type

C6686 (E) Invalid binary digit

C6688 (E) "this" pointer of "class name" is cast implicitly to near pointer

C6689 (E) Can not specify near or far for member

C6690 (E) A member "function name" qualified with near or far is declared

C6691 (E) Near or far specifier on a reference type is not allowed

C6692 (E) Can not specify near or far for member function

C6693 (E) Can not specify near or far for function types

C6694 (E) "this" pointer offset is too large

C6695 (E) Number of virtual function is too many

C6696 (W) Assertion warning

C6697 (I) Enumeration type is signed

Appendix G The SBDATA declaration & SPECIAL page Function declaration Utility (utl30)

How to startup the SBDATA declaration & SPECIAL page function declaration utility (utl30) and how the startup options works are described here.

G.1 Introduction of utl30

G.1.1 Introduction of utl30 processes

utl30 automatically puts highly frequently used variables or functions together in an SBDATA declaration or a SPECIAL page function declaration, thereby providing the optimization features for mapping those to an SB area or a special page area.
Figure G.1 shows the processing flow of utl30. First, compile and link all C/C++ source files using the compile option -finfo. An UTL file (extension ".utl") can be generated by adding the -utl option to optlnk at link time.
utl30 loads both the UTL file and the absolute file (extension ".abs") to generate a header file needed for optimization. Next, recompile and link all C/C++ source files. When recompiling, specify the header file generated by utl30 using the -preinclude option. Following the above procedure, it is possible to generate an optimized absolute file (.abs).

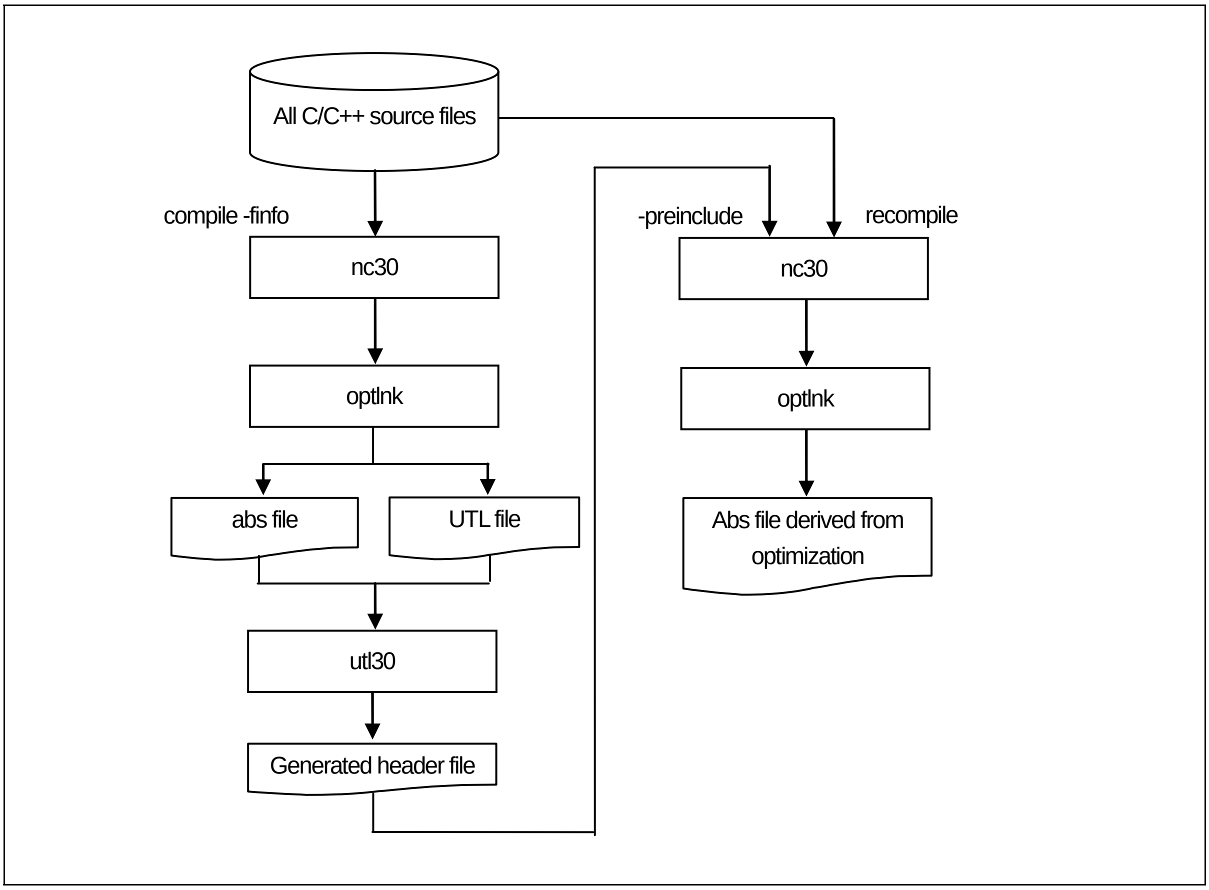


Figure G.1 Processing Flow of UTL30

G.2 Starting utl30

G.2.1 utl30 Command Line Format

For starting utl30, you have to specify the information and parameter that required.

```
% utl30△{ -sp30 | -sb30 } [△startup option]△absolute-file

%: Prompt
< >: Mandatory item
[ ]: Optional item
△: Space
Delimit multiple command line options with spaces.
```

Figure G.2 utl30 Command Line Format

To use utl30, specify the following for the startup option of this compiler

- output an inspector information.............................. -finfo option

An example for entering a command line is shown below.

```
% nc30 -finfo ncrt0.a30 sample.c
          Compile with -finfo option added.

% optlnk -utl -output=samle.abs    ncrt0.obj sample.obj
          Let the compiler generate abs and utl files.

% utl30 -sb30 -o samle sample.abs
          Let the compiler execute utl30 to generate a header. Do not add an extension for the output file name.

% nc30 -finfo -preinclude=sample.h ncrt0.a30 sample.c
          Specify a header with preinclude option before recompiling.

% optlnk -output=sample.abs ncrt0.obj sample.obj
          Generate an optimized abs file.
```

Figure G.3 Example of utl30 Command Input

G.2.2 Selecting Output Informations

To switch utl30 outputs between "SBDATA declaration" and "SPECIAL page function declaration," specify one of the options given below. If neither option is specified, an error results.

(1) Output SBDATA declaration
- Option "-sb30"

(2) Output SPECIAL page function declaration
- Option "-sp30"

G.2.3 Optional reference

-all

Output SBDATA declaration for all variables or SPECIAL page function declaration for all functions

Function :

- When used simultaneously with the -sb30 option
  Because the usage frequency is low, SBDATA declaration is output in the form of a comment for even the variables that are not placed in the SB area.
- When used simultaneously with the -sp30 option
  Because the usage frequency is low, SPECIAL declaration is output in the form of a comment for even the functions that are not placed in the SPECIAL page area.

Supplement : Use of this option helps to find the functions which are not called, even for once in program execution.
However, the functions which are called only indirectly require the user's attention, because such functions are indicated to have been called 0 times.

-fover_write -fOW

Outputs SBDATA declaration or SPECIAL function declaration to a file

Function : Does not check whether the output file specified by -o already exists. If such file exists, it is overwritten.
This option must be specified along with the -o option.

-fsection

Outputs SBDATA declaration and SPECIAL page function declaration in #pragma SECTIONS

Function : The variables and functions located in areas whose section names have been altered by #pragma SECTION are also included among those to be processed.

Notes : If #pragma SECTION is used for an explicit purpose of locating a particular variable or function at a given address, do not specify this option, because the variable or function may be located at an unintended different address by SBDATA or SPECIAL page declaration.

-o Output file name

Outputs the declared SBDATA result display file

Function : Outputs the result of SBDATA declaration or SPECIAL Page Function declaration to a file. With this option not specified, outputs the result to the host machine's(either EWS or personal computer) standard output device. If the specified file already exists, the result is written to the standard output device.
Do not add an extension for the output file name.
utl30 automatically adds the extension ".h" when it outputs a header.

-sb30

Outputs SBDATA declaration

Function : Outputs SBDATA declaration.
Specify the option -sb30 or the option -sp30.
If neither option is specified, an error results.

-sp30

Outputs SPECIAL page function declaration

Function : Outputs SPECIAL page function declaration.
Specify the option -sb30 or the option -sp30.
If neither option is specified, an error results.

-sp= <*number*>

Specifying numbers not be used as SPECIAL Page Function number option

Function : Specifies numbers not to be used as SPECIAL Page Function numbers.
Use it at the same time as option sp30.
When specifying more than one, use a comma to separate each or a hyphen to specify a range.
<Example format>

```
-sp=18,19        // Special page vector numbers 18 and 19 are unused.
-sp=200-255      // Special page vector numbers in the range 200 to 255 are unused.
```

-Wstdout

warning option

Function : Outputs error and warning messages to the host machine's standard output (stdout).

G.3 Notes

(1) In using utl30, .sbsym declared in files described in assembler cannot be counted. For this reason, you need to make adjustment, if a ".sbsym" declared in assembler is present, so that the results effected after having executed utl30 are put in the SB area.

(2) In using utl30, SPECIAL Page Function declared in files described in assembler cannot be counted. For this reason, you need to make adjustment, if a SPECIAL Page Function declared in assembler is present, so that the results effected after having executed utl30 are put in the SPECIAL Page area.

G.4 Conditions to establish SBDATA declaration & SPECIAL Page Function declaration

G.4.1 Conditions to establish SBDATA declaration

The variables that meet one of the following conditions are excluded from those to be processed.

- variables positioned in sections worked on by #pragma SECTION
- variables defined by #pragma ADDRESS
- const-qualified variables in cases where the compile option -fconst_not_ROM(-fCNR) is unused

If variables declared by use #pragma SBDATA have already been present in a program, the declaration is given a higher priority in using utl30, and variables to be allocated are picked out of the remainder of the SB area.

G.4.2 Conditions to establish SPECIAL Page Function declaration

The functions to be processed by utl30 are only those external functions that are listed below.

- Functions which are not declared with static
- Functions which are called four times or more

Note, however, that even the above functions may not be processed if they belong to one of the following:

- functions positioned in sections worked on by #pragma SECTION
- functions defined by any #pragma

If variables declared by use #pragma SPECIAL have already been present in a program, the declaration is given a higher priority in using utl30, and variables to be allocated are picked out of the remainder of the SPECIAL page area.

G.5 Example of utl30 use

G.5.1 Generating a SBDATA declaration file

a. Generating a SBDATA declaration file

Adding the -sb30 option to utl30, it is possible to output an SBDATA declaration file.

```
/*
* #pragma SBDATA Utility
*/
/* SBDATA Size [256] */
#pragma   SBDATA   gi                      /* size=(    2) / ref=[       2] / gi */
/*                                            (A)           (B)           (C)
* End of File
*/

(A) Shows size of a data.
(B) Shows access count of the variables.
(C) Shows a name in the source file.
```

Figure G.4 Example of a Header Generated when -sb30 Option is Specified

b. Adjustment in an instance in which SB declaration is made in assembler

If external variables are defined with the assembler directive ".sbsym," the header files generated by utl30 need to be adjusted.

Assembly language Program

```
        .sbsym      _sym
          :
        (omitted)
          :
        .glb        _sym
_sym:
        .blkb       2
```

Generated file by utl30

```
/*
 * #pragma SBDATA Utility
 */
/* SBDATA Size [255] */
#pragma SBDATA      data3           /*  size = (4) / ref = [2] / data3   */
#pragma SBDATA      data2           /*  size = (1) / ref = [1] / data2   */
          :
        (omitted)
          :
#pragma SBDATA      data1           /*  size = (2) / ref = [1] / data1   */
/*
 * End of File
 */
```

Since 2-byte data are SB-declared in an assembler routine,you subtract 2 bytes of SBDATA declaration from the file generated by utl30.

Example)

```
          :
        (omitted)
          :
//#pragma SBDATA    data1           /*  size = (2) / ref = [1] / data1   */
/* Comments out*/
```

Figure G.5 Example of adjust the file generated by utl30

G.5.2 Generating a SPECIAL Page Function declaration file

a. Generating a SPECIAL Page Function declaration file

Adding the -sp30 option to utl30, it is possible to output a SPECIAL page function declaration file.

```
/*
 *      #pragma SPECIAL PAGE Utility
 */
/*      special page definition */
#pragma   SPECIAL   255   func()                 /* size=(     6) / ref=[        4] / func() */
/*                                                 (A)             (B)             (C)
 *      End of File
 */
(A ) Indicates the function size.
(B) Indicates the reference frequency of function.
(C) Indicates a name in the source file.
```

Figure G.6 Example of a Header Generated when -sp30 Option is Specified

G.6 utl30 Error Messages

G.6.1 Error Messages

Table G.1, Table G.2 lists the utl30 calculation utility error messages and their countermeasures.

Table G.1 utl30 Error Message List (1/2)

Error No.	Type	Message	Content of error and solution
U2100	Error	Illegal file extension 'extension'	• The extension of the input file is not "abs." ⇒Check the input file.
U2101	Error	Illegal file extension ' '	• The input file does not have an extension. ⇒Specify a correct file name.
U2200	Error	ignore option 'input option name'	• An invalid option is input. ⇒Check the option.
U2201	Error	ignore option '-sp'	• Selecting the -sp option while the -sb30 option is selected. ⇒-The -sp option can be specified simultaneously with the -sp30 option.
U2301	Error	Option '-sp' is not appropriate	• The specified -sp option contains a character other than numeric values.
U2402	Error	No input 'abs' file specified	• The selected input file is not an abs file. Or unable to load an abs file.
U2403	Error	No input 'input abs filename' file specified	• Unable to load an abs file.
U2600	Error	'-SB30/-SP30' is missing	• Neither the -sb30 nor the -sp30 option is selected. ⇒Specify either one.
U2700	Error	cannot open 'input utl filename' file	• Unable to open the utl file.
U2701	Error	cannot read header file 'input abs filename'	• The abs file is erroneous. It may be corrupted.
U2702	Error	cannot read symbol table	• The abs file is erroneous. It may be corrupted.
U2703	Error	cannot read section header	• The abs file is erroneous. It may be corrupted.
U2704	Error	cannot read section data	• The abs file is erroneous. It may be corrupted.
U2705	Error	cannot read ELF header	• The abs file is erroneous. It may be corrupted.
U2706	Error	cannot open output file 'output filename.h'	• Unable to open the output file.
U2707	Error	cannot close file 'output filename.h'	• Unable to close the output file.
U2800	Error	Illegal File Format 'input utl filename' file	• The utl file is erroneous. It may be corrupted.
U2801	Error	Illegal File Format 'input abs filename'	• The abs file is erroneous. It may be corrupted.
U2802	Error	Illegal file format	• The abs file is erroneous. It may be corrupted.
U2900	Error	not enough memory	• Memory is insufficient. Close the unnecessary files.

Table G.2 utl30 Error Message List (2/2)

Error No.	Type	Message	Content of error and solution
U0001	Information	Since 'output filename.h' file exists, it makes a standard output.	• A file selected with the -o option already exists.
U1000	Warning	warning: conflict declare of variable name	• The variable concerned is declared with different storage classes, types, etc. between different files.
U1001	Warning	warning: conflict declare of function name	• The function concerned is declared with different storage classes, types, etc. between different files.

Appendix H Library Generator

The library generator (lbg30) is a tool that creates standard library files (.lib) conforming to the options specified by the user.

To link a library using the standard library created by lbg30, specify the library file as shown below.
$ nc30 -fno_lib -l generated standard library • • •

To specify from the linker,
$ optlnk -library=generated standard library • • •

H.1 Command Line Syntax

% lbg30 [△option1][△option2]

- For option1, specify one of the options described in H.3, "Library Generator Options."
- For option2, specify one of the options described in H.4, "Compiler Options Specifiable for the Library Generator."

Example: lbg30 -output=mylib.lib -head=stdio -exception -rtti=on

H.2 Precautions to Take When Using lbg30

The library generator invokes the nc30 compiler, so be sure that the environment variables required for the compiler to run have already been set before the invocation.
The library generator uses the following folders:

- Folders specified by the environment variable TMP30
- Current directory

Since the library generator writes to these folders, be sure that the folders are write-enabled.

H.3 Library Generator Options

The options specifiable for the command-line option "option1" of the library generator are listed below.

Table H.1 List of Library Generator Options

Option	Content
-head=<sub>[,...]	Specifies the target library to be built
<sub>:{ all	All library functions and runtime libraries
| runtime	Runtime library
| ctype	ctype.h (C89) and runtime library
| math	math.h (C89) and runtime library
| mathf	mathf.h (C89) and runtime library
| stdarg	stdarg.h (C89) and runtime library
| stdio	stdio.h (C89) and runtime library
| stdlib	stdlib.h (C89) and runtime library
| string	string.h (C89) and runtime library
| ios	ios (EC++) and runtime library
| new	new (EC++) and runtime library
| complex	complex (EC++) and runtime library
| cppstring	string (EC++) and runtime library
}	
-output=<file name>	Specifies output library file name
-nofloat	Generates simplified I/O function

-head

Synopsis -head=<sub>[,…]

<sub>:{ all

| runtime | ctype | math | mathf | stdarg | stdio | stdlib | string | ios | new | complex | cppstring

}

Description Specifies the target library to build by a header file name.

When -head=all is specified, all header files are assumed to be the target to build.

The runtime library is always the target to build.

The default of this option is -head=all.

-output

Synopsis -output=<filename>

Description Specifies the output file name.

The default of this option is -output=stdlib.lib.

-nofloat

Synopsis -nofloat

Description Generates a simplified I/O function that does not support floating-point conversions (%f, %e, %E, %g, and %G). When file input/output that does not require floating-point conversions is performed, the code size can be reduced.

Subject functions fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf

Remarks For a library created after specifying this option, if floating-point numbers are input or output in the subject function, behavior is not guaranteed.

H.4 Compiler Options Specifiable for the Library Generator

The options specifiable for the command-line option "option2" of the library generator are listed below.

Table H.2 List of Compiler Options Specifiable for the Library Generator

No.	Option
1	-exception
2	-noexception
3	-rtti=on
4	-rtti=off
5	-O
6	-O1
7	-O2
8	-O3
9	-O4
10	-OR
11	-OS
12	-fdouble_32 (-fD32)
13	-fptrdifft_16 (-fP16)
14	-fsizet_16 (-fS16)
15	-R8C
16	-R8CE
17	-goptimize
18	-fno_align (-fNA)
19	-Ostack_frame_align (-OSFA)

Appendix I C Language Behavior Under NC30

With regard to the "undefined behavior," "implementation-defined behavior," and "locale-specific behavior" stipulated in ANSI standards, this chapter describes behavior in C language under the C compiler NC30. These behaviors are explained corresponding to the respective sections in ANSI standard ANSI/ISO 9899—1990.
In ANSI standards, the "undefined behavior," "implementation-defined behavior," and "locale-specific behavior" are defined as follows:

a. Undefined behavior

Behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which this International Standard imposes no requirements.

b. Implementation-defined behavior

Behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.

c. Locale-specific behavior

Behavior that depends on local conventions of nationality, culture, and language that each implementation shall document.

I.1 Undefined Behavior in ANSI Standards

Actions handled as "undefined behavior" in ANSI standards are not guaranteed to be processed normally by a C compiler. In most cases, they are ignored, alerted by an error message or warning, or result in runtime error. Therefore, coding that will not come under the category "undefined behavior" is recommended.

Shown below, beginning with the next paragraph, are the predicted (though not guaranteed) behavior in the C compiler NC30 of actions handled as "undefined behavior." The numbers and headings following "■ ANSI standard" denote the corresponding section numbers and section titles in ANSI standard "ANSI/ISO 9899—1990."

■ ANSI standard 5.1.1.2, Translation phases (end of the source file)
If the source file has no new-line character at the bottom of it, a new-line character is automatically added. (The last line of a file does not need to end in a new-line character.)
If the source file ends in a new-line character preceded by a backslash, the backslash and new-line character are deleted, and two instances of new-line character are added. If the source file ends in a preprocessing token [Note 1] or in the middle of a comment statement, an error is assumed.

Note 1: A preprocessing token (see ANSI standard 6.1 for details) is the basic processing unit of text in a C source file. It includes: header filenames, identifiers, preprocessing numbers, character constants, string literals, operators, punctuators, and single non-white-space characters other than those mentioned above.

■ ANSI standard 5.2.1, Character sets (characters other than the character sets)
If any character other than the character sets usable in source files (except the preprocessing tokens not converted into tokens, character constants, string literals, header names, and comment statements) occurs in the source file, an error is assumed.

■ ANSI standard 5.2.1.2, Multibyte characters
If a multibyte character is used in other than comments, character constants, or string literals, the behavior of codes generated by this compiler cannot be guaranteed. Also, if the end of a comment "*/" is immediately preceded by a shift state (other than the initial shift state), the end of the comment may not be recognized.

■ ANSI standard 6.1, Lexical elements (pair of quotes)
If a quote ' or " that is not paired in the source occurs, an error is assumed.

■ ANSI standard 6.1.2.1, Scopes of identifiers
If the same identifier is used as label two times or more in one and the same function, an error is assumed.
If any identifier nonexistent in the current scope is used, the behavior of generated code cannot be guaranteed.

■ ANSI standard 6.1.2, Identifiers
As for identifiers designating the same thing, if their constituent characters following the significant digits differ, they are not guaranteed.

■ ANSI standard 6.1.2.2, Linkages of identifiers
If the same identifier denoting a function is declared on both internal linkage and external linkage sides, the behavior of generated code cannot be guaranteed.

■ ANSI standard 6.1.2.4, Storage durations of objects
If, while storage reserved for an object with automatic storage duration is no longer guaranteed, a pointer value referring to that object is used, behavior of the program cannot be guaranteed, although no compile error results.

■ ANSI standard 6.1.2.6, Compatible type and composite type
If there are two declarations for the same object or function and their types are not compatible, the behavior of generated code cannot be guaranteed.

■ ANSI standard 6.1.3.4, Character constants
If an unsupported escape sequence occurs in character constants or string literals, an error is assumed. (Example: '\C' results in error.)

■ ANSI standard 6.1.4, String literals
If a character string literal and a wide string literal exist next to each other, they are concatenated simply as they are, without being adjusted to the respective types.

■ ANSI standard 6.1.7, Header names
Even if the characters \,. " ", or /* occur in a header name, they are recognized as the characters comprising a file name (not processed as special characters).

■ ANSI standard 6.2.1, Arithmetic operands
If an arithmetic conversion produces a result that is not representable in a given space (insufficient precision), an approximate value is taken. For conversions to integers, however, the digits below the decimal point and the bit patters of the high-order digits that do not fit in the space are discarded.

■ ANSI standard 6.2.2.1, Lvalues

Use of an incomplete type for lvalue, except when initializing arrays in an initialization expression, results in an error.

■ ANSI standard 6.2.2.2, void

If an attempt is made to use a value with type void for access or apply an implicit conversion to a void expression (except for conversions to void), an error is assumed.

■ ANSI standard 6.3, Expressions

Side effect

Side effects produced between sequence points of an expression are indeterminate. Do not write code that is likely to yield different results due to side effects.

For example, the code "*p++=*p+5" may be evaluated in different ways, with *p+5 evaluated prior to p++ in one or after p++ in the other, so that the destination in which *p+5 is to be stored becomes indeterminate. In this case, either one of the coding given below should be followed, depending on the desired processing.

```
*p=*p+5;
++p;
```

or

```
*p = *(p+1)+5;
p++;
```

Invalid arithmetic, domain error

For invalid arithmetic operations (e.g., division by 0), and for the case where an operation results in a domain error (e.g., overflow or underflow), the behavior of generated code cannot be guaranteed.

■ ANSI standard 6.3.2.2, Function calls

When arguments to functions are a void expression

Specifying a void expression other than null parameters as argument to a function results in an error. Also, if, where a null parameter (void expression) is specified, more than one parameter is defined for the called function, the value passed to the function is indeterminate.

When argument and parameter types do not match

In a function call where function prototype declarations are nonexistent, if the function is defined in a place invisible to the function declaration and the promoted (implicitly converted) argument and the parameter do not have matching type, the value of the argument cannot be guaranteed.

When function prototypes and function definitions differ in type

For a function call where function prototype declaration is visible but the defined type of the function is not compatible with its declaration, the behavior of generated code cannot be guaranteed.

Prototype declarations with variable arguments

If a function that accepts a sequence of variable arguments is called in a place where a function prototype terminating with "..." is outside the function prototype scope, the behavior of generated code cannot be guaranteed.

■ ANSI standard 6.3.3.2, Unary operators (&, *)

If one of the following references is attempted by means of the address operator & and the indirection operator *, behavior cannot be guaranteed.

- References to invalid arrays
- References to NULL points
- References to objects that have an automatic storage duration whose scope has terminated

■ ANSI standard 6.3.4, Cast operators

If a pointer to one function is cast to a pointer to another function of different type and a function that has a type incompatible with the original type is called, behavior cannot be guaranteed.
If pointers are cast to integer type (including character type) or cast to other than pointer type, such an attempt often results in an error. Note also that even if no error is assumed, behavior of the program cannot be guaranteed.

■ ANSI standard 6.3.6, Additive operators

Even when a pointer to an array is added and/or subtracted and the operation results in the pointer indicating other than the array-element area, no compile error is assumed. In this case, although the content pointed to by the pointer can be referenced with the * operator, because this data is not an array element, behavior of the program cannot be guaranteed.

■ ANSI standard 6.3.7, Shift operators

If the specified amount of shift in a shift operation is negative or exceeds the bit width of a shifted expression, behavior cannot be guaranteed. (Example: In cases when the specified amount of shift is negative, the shift direction may be reversed. When the specified amount of shift exceeds the bit width of a shifted expression, the expression may be shifted normally, as long as it is representable by size of its type.)

■ ANSI standard 6.3.8, Relational operators

Even if pointers compared by a relational operator (<, <=, >, or >=) do not point to objects included in the same aggregate (structure or array), no error is assumed but behavior cannot be guaranteed.

■ ANSI standard 6.3.16.1, Assignment operators (simple assignment =)

If an object is assigned to an overlapping object, the behavior of generated code cannot be guaranteed.

■ ANSI standard 6.5, Declarations

If an object declared without linkage—even after its declaration has terminated, or after its initial declaration has terminated (providing the object has an initial value)—is of incomplete type, an error is assumed.

■ ANSI standard 6.5.1, Storage-class specifiers

If a function is declared with other than the extern storage-class specifier in a block scope, behavior cannot be guaranteed.

■ ANSI standard 6.5.2.1, Structure and union specifiers

Unnamed members

If a structure or union consisting only of unnamed members is defined, behavior cannot be guaranteed.

Bit-field types of structures

The valid types usable in the bit-field declarations of structures are signed or unsigned char, short, int, long, and _Bool. If any other types are declared, behavior cannot be guaranteed.

■ ANSI standard 6.5.3, Type qualifiers

If an attempt is made to modify an object declared as const by an lvalue other than const—in other words, if an attempt is made to process a const-declared area as if it were not const by means of a cast, etc., behavior cannot be guaranteed (in some cases, no error is assumed).

If an attempt is made to modify an object declared as volatile by an lvalue other than volatile—in other words, if an attempt is made to process a volatile-declared area as if it were not volatile by means of a cast, etc., behavior cannot be guaranteed (in some cases, no error is assumed).

■ ANSI standard 6.5.7, Initialization

If an uninitialized object that has automatic storage duration is used before it is assigned a value, its value is indeterminate.

■ ANSI standard 6.6.6.4, The return statement

If a function value is referenced, but the value is not return'ed on the function side, the referenced function value is indeterminate.

■ ANSI standard 6.7, External definitions

When two or more instances of the same identifier that has external linkage are defined, an error results at compile time if they exist in one and the same source, or an error results at link time if they exist sporadically in multiple sources.

However, if function/external variable definitions and variables exist sporadically in multiple sources under the following conditions, no errors may be assumed.

- Functions
  - Function prototype and K&R coexist.
  - Parameters are different.
- Variables
  - Types are different.

■ ANSI standard 6.7.1, Function definitions

If, where there is a function that accepts variable arguments and the parameter list of its function definition does not terminate with "...", a greater number of arguments than declared in the parameter list are passed to the function, behavior cannot be guaranteed.

■ ANSI standard 6.7.2, External object definitions

If the identifier of an object of incomplete type that has internal linkage is declared by an ambiguous definition, behavior cannot be guaranteed (in some cases, a warning results).

■ ANSI standard 6.8.1, Conditional inclusion

The tokens defined generated during expansion of preprocessing directives #if or #elif are handled as operators.

■ ANSI standard 6.8.2, Source file inclusion

If a #include preprocessing directive does not conform to any one of the forms below, an error is assumed.

- <file name>
- "file name"

■ ANSI standard 6.8.3, Macro replacement

A function-like macro invocation without arguments results in an error.

If a line beginning with #, or a preprocessing directive, exists in the argument list of a macro call, it is considered to be a preprocessing directive.

■ ANSI standard 6.8.3.2, The # operator (turns into a string)
If an operation to turn a line into a string by the preprocessing operator # does not result in a valid string constant, behavior cannot be guaranteed. An error may result at expansion time.

■ ANSI standard 6.8.3.3, The ## operator (connects tokens)
If an operation to connect tokens by the preprocessing operator ## does not result in a valid preprocessing token, behavior cannot be guaranteed. For example, func##1 becomes func1 when it is expanded, but if func1 is an meaningless token, a warning may result at compile time or an error may result at link time.

■ ANSI standard 6.8.4, Line control (#line)
If a #line preprocessing directive after being expanded does not conform to grammar, an error is assumed. In this case, line information is not updated.

■ ANSI standard 6.8.8, Predefined macro names
The names __LINE__, __FILE__, __DATE__, and __TIME__ are predefined macros, so that an attempt to define any of these or cancel their definitions by #define or #undef results in a warning.

■ ANSI standard 7, Library
If an attempt is made to copy an object to an overlapping object by using any library function other than memmove, the data in overlapping part is not guaranteed.

■ ANSI standard 7.1.2, Standard headers

<u>Include within an external definition</u>
The function declarations, object declarations, type definitions, and macro definitions supplied with the C standard library, as well as macro definitions that have the same names as keywords, require that the corresponding standard header files be included before they are referenced first. If included after being referenced, they will not work correctly.

<u>Redefinition of reserved external names</u>
If the external names reserved for a program (e.g., external names in headers) are defined, how they will be processed depends on link order.

■ ANSI standard 7.14, Errors <errno.h>
The errno is defined with external variables.

■ ANSI standard 7.1.6, Common definitions <stddef.h>
Specifying bit-field members in structure form for the second parameter of an offsetof macro results in an error.

■ ANSI standard 7.1.7, Use of library functions
If an argument to any library function is an invalid value, behavior of the program cannot be guaranteed.
For library functions that accept variable arguments, unless they are declared by a header inclusion, etc., the function concerned may not work correctly.

■ ANSI standard 7.2, Diagnostics headers <assert.h>
The assert is implemented by a macro. If an assert is called by suppressing a macro invocation in order to access a function, a warning results at compile time and, because external symbols are nonexistent, an error results at link time.

■ ANSI standard 7.3, Character handling function headers <ctype.h>

If an argument to a character handling function is representable by unsigned char or not equal to the value of a macro EOF, behavior cannot be guaranteed.

■ ANSI standard 7.6, Nonlocal jump function headers <setjmp.h>

The setjmp is such that even if a macro definition is suppressed, no error is assumed.

■ ANSI standard 7.6.1.1, The setjmp macro

The setjmp macro is recommended to be used for the purposes mentioned below. Although no error is assumed even if it is used for other than those purposes, should it be used in a complicated expression, part of the current execution environment (e.g., intermediate result of the evaluation of an expression) may be lost.

- Operand control in the comparison of selection statements, iteration statements, and integer constant expressions (e.g., implicit processing by the unary operator !)
- Operand control of selection statements or iteration statements
- Expression statements (e.g., cast to void)

■ ANSI standard 7.6.2.1, The longjmp function

If an object of automatic storage class that is not specified as volatile is altered during an interval from setjmp execution to longjmp invocation, the value of the object cannot be guaranteed.

■ ANSI standard 7.7.1.1, The signal function

The C standard library of this compiler does not have the signal function implemented in it.

■ ANSI standard 7.8.1, Variable argument access function headers <stdarg.h>

Assuming some function (let it be a function A) and another one that is called with ap (sequence of variable parameters) of a va_arg macro as arguments to it (let it be a function B), wherein if the va_arg macro is invoked using this ap, a reference to arguments becomes as follows:

- On the function B side (one that was called from the function A), it is possible to refer from variable arguments pointed to by ap at the time it was called.
- On the function A side (one that called the function B), it is possible to refer from variable arguments pointed to by ap at the time it called the function B, regardless of whether the function B refers to the variable arguments.

However, if the address of ap is passed to as argument, or an aggregate (if ap is an aggregate) is passed to as argument, then ap of the function A after return from the function B is a sequel from where the function B has terminated.

The va_start, va_arg, and va_end are implemented by a macro. If they are called after suppressing macro invocation in order to access a function, an error results at link time because external symbols are nonexistent.

■ ANSI standard 7.8.1.1, The va_start macro

If the declared type of the second parameter to the va_start macro is a register class variable, function type, or an array type or does not match the type of parameter after default argument promotion (type after implicit conversion on parameters), behavior cannot be guaranteed.

■ ANSI standard 7.8.1.2, The va_arg macro

If, when va_arg is invoked, the argument to be processed does not actually exist, behavior cannot be guaranteed.

If, when va_arg is invoked, the argument to be processed is not of a specified type, behavior cannot be guaranteed.

■ ANSI standard 7.8.1.3, The va_end macro

Even if va_end is invoked before invocation of the va_start macro, no error is assumed and it works normally.

Even if a function that has a variable argument list initialized by the va_start macro returns before invocation of the va_end macro, no error is assumed, but behavior of the program cannot be guaranteed.

■ ANSI standard 7.9.5.2, The fflush function

This function returns 0 without performing any operation.

■ ANSI standard 7.9.5.3, The fopen function

The C standard library of the M16C C compiler does not have the fopen function implemented in it.

■ ANSI standard 7.9.6, Formatted input/output functions

<u>pintf and scanf series</u>

If type in function specification and the corresponding number in the argument list do not match, and if the number of conversion specifiers is smaller than that of arguments, behavior cannot be guaranteed, although no error is assumed. If the number of arguments is larger than that of conversion specifiers, the excess arguments are ignored.

Input/output results for invalid conversion specification in a pintf or scanf-series function are indeterminate. In most cases, no error message is output. If input/outputs are not obtained as expected, please check to see if the conversion specification is coded in the correct form.

<u>%% conversions in pintf and scanf series</u>

In the conversion specification %% of a pintf or scanf-series function, the character next to % is processed as a conversion specifier.

■ ANSI standard 7.9.6.1, printf series

<u>Qualifiers</u>

If, in printf-series conversion specification, a qualifier (size specifying character h, l) is specified prior to a h or l preceding any conversion specifier other than the relevant conversion specifier (o, x, X, e, E, f, g, G), the qualifier is ignored.

<u>Flags</u>

If, in printf-series conversion specification, a flag # is specified prior to other than the relevant conversion specifier (o, x, X, e, E, f, g, G), the flag is ignored.

If, in printf-series conversion specification, a flag 0 is specified prior to other than the relevant conversion specifier (d, i, o, u, x, X, e, E, f, g, G), the flag is ignored.

<u>Conversion result</u>

If an aggregate, a union, or a pointer to an aggregate or union is specified for other than printf-series %p and %s, behavior cannot be guaranteed. If a single % conversion by a printf function results in character output exceeding 30 characters, behavior cannot be guaranteed.

■ ANSI standard 7.9.6.2, scanf series

Qualifiers

If, in printf-series conversion specification, a qualifier (size specifying character h, l, L) is specified prior to other than the relevant conversion specifier as shown below, the qualifier is ignored.

- h or l preceding any conversion specifier other than d, i, n, o, u, or x
- L preceding any conversion specifier other than e, f, or g

Compatibility with printf-series %p

The output form of printf-series %p conversions and the address form that is stored in a scanf-series %p are compatible with each other.

Conversion result store area

If the area in which the result of a conversion by a scanf-series function is stored lacks in size or has an incompatible type, behavior cannot be guaranteed.

■ ANSI standard 7.10.1, String conversion functions (conversion from string to numeric value)

If the result of a conversion by a string-to-numeric value conversion function (atof, atoi, or atol) is an unrepresentable value due to a domain error, the minimum value or maximum value is returned. In this case, ERANGE is set in errno.

■ ANSI standard 7.10.3, Memory management functions (free, realloc)

If an area released by a free or realloc function is referenced, behavior cannot be guaranteed.

If one of the following values is passed to a free or realloc function as its first argument (pointer to the area to be released), behavior cannot be guaranteed.

- Any value other than the return value of a calloc, malloc, or realloc function (pointer to an allocation-completed area)
- A pointer to the area previously released by a free or realloc function

■ ANSI standard 7.10.4.3, The exit function

If a program executes a call to the exit function more than once, its behavior cannot be guaranteed. An infinite loop is entered into by a first call of the exit function.

■ ANSI standard 7.10.6, Integer arithmetic functions

If the result of an integer arithmetic function (abs, div, labs, or ldiv) is not representable, its value cannot be guaranteed.

■ ANSI standard 7.10.7, Multibyte character functions (shift state)

Since only the C locale is supported, locales cannot be changed.

■ ANSI standard 7.11.2 and 7.11.3, Copying and concatenation functions

If one of the following cases applies, behavior cannot be guaranteed.

- In memcpy, memmove, strcpy, and strncpy functions, the size of the destination to which copied is smaller than that of the source from which copied.
- In strcat and strncat functions, the area reserved for strings to be concatenated is insufficient to store the concatenated string.

■ ANSI standard 7.12.3.5, The strftime function

The C standard library of the M16C C compiler does not have the strftime function implemented in it.

I.2 Implementation-Defined Behavior

With regard to the actions handled as "implementation-defined behavior" in ANSI standards, the following describes behavior in C language under the C compiler NC30. Mainly, the manner in which error messages are notified, the number of characters valid as an identifier, and the integer and floating-point formats are stipulated

The numbers and headings following "■ ANSI standard" denote the corresponding section numbers and section titles in ANSI standard "ANSI/ISO 9899—1990." The items handled as "implementation-defined behavior" are enclosed in angle brackets <　>, and the corresponding operation of NC30 are explained after the brackets.

I.2.1 Translation

■ ANSI standard 5.1.1.3, Diagnostics

<Message output form of the C compiler>

The diagnostic messages of the C compiler consist of warning messages, error messages, and serious error messages. For message output forms and other details, please see Appendix F, "Error Message List."

I.2.2 Environment

■ ANSI standard 5.1.2.2.1, Program startup

<Meanings of arguments to the main function>

The arguments passed to the main function depend on specifications of the startup program created by the user.

■ ANSI standard 5.1.2.3, Program execution

<One that comprises an interactive device>

The behavior of input/output devices depends on specifications of the low-level functions created by the user.

I.2.3 Identifiers

■ ANSI standard 6.1.2, Identifiers

<For identifiers without external linkage, how many characters from the beginning of a string (exceeding 31) are recognizable>

For identifiers that do not have external linkage, up to 255 characters from the beginning of a string are valid. The 256th character and those that follow are ignored.

<For identifiers with external linkage, how many characters from the beginning of a string (exceeding 6) are recognizable>

For external identifiers, up to 255 characters from the beginning of a string are valid. The 256th character and those that follow are ignored. Also, external identifiers are case-sensitive.

I.2.4 Characters

■ ANSI standard 5.2.1, Character sets

<Kinds of source and execution character sets (not including those that are explicitly specified in the International Standard)>

Both source character set and execution character set can be realized using the characters defined in JIS X0201 and 0208. However, the Latin alphabet part of JIS X0201 are considered as ASCII when they are processed. For the actual character code (encode), EUC (Expanded Unix Code) and Shift JIS can be used.

■ ANSI standard 5.2.1.2, Multibyte characters

<Shift state of multibyte characters>

For multibyte characters, there are no shift states (the strings indicating the beginning and end of multibyte characters).

■ ANSI standard 5.2.4.2.1, Sizes of integer types

<Number of bits comprising one character of the execution character set>

One character of the execution character set consists of 8 bits.

■ ANSI standard 6.1.3.4, Character constants

<Mapping of the source character set to the execution character set>

The characters in the source character set are mapped one-for-one to the execution character set.

<Values of integer character constants that include the characters nonexistent in the basic execution character set or in the extended character set of wide character constants>

They are derived by concatenating the two leftmost characters in big-endian mode.

<Values of integer character constants consisting of more than one character or wide character constants consisting of more than one multibyte character>

The character constants that are not wide characters assume the value of the leftmost character. The values of wide character constants depend on the environment variable, NCKOUT.

<Locales needed for converting multibyte characters to the corresponding wide characters (code)>

No locales but the "C" are supported.

■ ANSI standard 6.2.1.1, Characters and integers

<To which is a char akin, signed char or unsigned char>

A char, in its generated code, behaves the same way as unsigned char.

I.2.5 Integers

- ANSI standard 6.1.2.5, Types

<Representation of integer types>

For the internal representation and limit values of various integer type data, please see Appendix D.1, "Internal Representation of Data." Note, however, that the C compiler interprets int and singed int, short and singed short, and long and signed long as being the same one, respectively.

- ANSI standard 6.2.1.2, Singed and unsigned integers

<In cases where values are unrepresentable, the result derived by converting integer data to a signed integer type shorter than that, or the result derived by converting unsigned integer data to a signed integer type of the same size>

When an integer is converted to a "signed integer smaller in size than that," the lower-bit value of the original integer is converted to a signed integer directly as is. The most significant bit of the converted signed integer is a sign bit.

When an unsigned integer is converted to a "signed integer of the same size," the lower-bit value of the original integer is converted to a signed integer directly as is.

- ANSI standard 6.3, Expressions

<Results of bitwise operations of signed integers>

The bitwise operations of signed integers are handled as though they are unsigned integers.

- ANSI standard 6.3.5, Multiplicative operators

<Signs of remainders resulting from divisions of integers>

The remainders assume the same sign as that of the dividend.

- ANSI standard 6.3.7, Bitwise shift operators

<Right shift of signed integer types that have negative values>

The right shift of signed integer types is an arithmetic shift.

I.2.6 Floating Point

■ ANSI standard 6.1.2.5, Types

<Representation of floating types>

For the internal representation and limit values of various floating type data, please see Appendix D.1.2, "Floating Types."

■ ANSI standard 6.2.1.3, Floating and integral

<Rounding mode when an integer is converted to a floating type and the original numerical value cannot be represented exactly>

Rounded to a value nearest to the original value of the integer within the range representable by the floating type to which it was converted.

■ ANSI standard 6.2.1.4, Floating types

<Rounding mode when a floating type is converted to another floating type of a smaller size>

Rounded to a value nearest to the original value of the floating type within the range representable by the other floating type to which it was converted.

I.2.7 Arrays and Pointers

■ ANSI standard 6.3.3.4, The sizeof operator, and 7.1.1, Definition of library terms

<Type size_t of the sizeof operator>

Type size_t of the sizeof operator is defined by default as unsigned long, while when the compile option -fsizet_16(-fS16) is specified, it is defined as unsigned int.

■ ANSI standard 6.3.4, Cast operators

<Cast of pointer type to integer type and vice versa>

When pointers are converted to integers or integers are converted to pointers, the pointer is assumed to be an unsigned integer when a conversion is performed.

If the number of bits in a type before being converted and that in a converted type are the same, their bit patterns are used directly as are.

If a converted type has a smaller number of bits, as many bits as in the converted type are used, beginning with the least significant bit.

If a converted type has a larger number of bits, a pointer-to-integer conversion is zero-extended, a signed integer-to-pointer conversion is signed-extended, and an unsigned integer-to-pointer conversion is sign-extended. The low-order bit pattern corresponding to the number of bits before being converted does not change.

■ ANSI standard 6.3.6, Additive operators, and 7.1.1, Definition of library terms

<Type of ptrdiff_t>

Type ptrdiff_t of integers that hold the difference between two pointers is defined by default as signed long, while when the compile option -fptrdifft_16(-fP16) is specified, it is defined as signed int.

I.2.8 Registers

■ ANSI standard 6.5.1, Storage-class specifiers

<Number of objects that can be declared as register>

There are no limits to the number of objects that can be declared as register.

By default, the storage-class specifier register is ignored.

When the compile option -fenable_register(-fER) is specified, variables of integer type or pointer type in size of 32 bits or less that are declared as register are mapped to registers when accessed.

If they cannot be mapped to registers at the same time, part of objects mapped to registers is temporarily saved to the stack.

I.2.9 Structures, Unions, Enumerators, and Bit-fields

■ ANSI standard 6.3.2.3, Structure and union members

<Where union members are accessed by a member of different type>

The bit patterns stored in union members are accessed, whose value is interpreted according to the type of the accessed member.

■ ANSI standard 6.5.2.1, Structure and union specifiers

<Padding and alignment of structure members>

For details about the padding and alignment of bit-fields, please see Appendix D.1, "Internal Representation of Data."

<Whether bit-fields of type "int" are a "signed int" or a "unsigned int">

The bit-fields, not explicitly indicated to be signed or unsigned, are handled as unsigned.

<The order in which bit-fields are mapped to storage device>

Bit-fields are mapped in ascending order of bits, from the low-order bit to higher-order bits.

<Whether bit-fields overlap storage boundaries>

In no event will one bit-field overlap a storage boundary when they are mapped.

Storage for bit-fields are created in units equal to the number of bits that their declared type will have when they are not a bit-field (e.g., 8 bits for char, or 16 bits for int).

■ ANSI standard 6.5.2.2, Enumeration specifiers

<Type of the values of enumerated types>

By default, enumerated types are handled as the one that is compatible with type unsigned int.

When the compile option -fchar_enumerator(-fCE) is specified, enumeration type is handled as the one that is compatible with type unsigned char.

I.2.10 Qualifiers

- ANSI standard 6.5.3, Type qualifiers

<Method for accessing objects with volatile qualifier>
Each time a volatile object name is referenced, the object is accessed. No optimizations are performed on volatile objects.

I.2.11 Declarators

- ANSI standard 6.5.4, Declarators

<Maximum number of declarators where types of arithmetic operations, structures, and unions are correctable>
There are no particular limits to the maximum number of declarators.

I.2.12 Statements

- ANSI standard 6.6.4.2, The switch statement

<Maximum number of case values in a switch statement>
The maximum number of case values in a switch statement depends on the usable memory capacity of the host machine.

I.2.13 Preprocessing Directives

- ANSI standard 6.8.1, Conditional inclusion

<Whether the value of a single-character character constant in a constant expression that controls conditional inclusion will match that of the same character constant in the execution character set, or whether such a character constant will have a negative value>
The value of a single-character constant in a constant expression that controls conditional inclusion matches that of the same character constant in the execution character set. All of such a character constant have an unsigned positive value.

- ANSI standard 6.8.2, Source file inclusion

<Method of searching for include files (header files)>
The header files specified by #include are searched in the order mentioned below.
For the header files enclosed in < >

(1) The directory specified by the startup option -I of NC30
(2) The standard directory that is set by the environment variable, INC30

For the header files enclosed in " "

(1) The directory that contains the source file
(2) The directory specified by the startup option -I
(3) The standard directory that is set by the environment variable, INC30

<Include file names enclosed in quotes>
The #include preprocessing directive also permits use of quotes in specifying an include file name.

<Mapping of source file character sequence>
The characters in a source file are assigned the values of the respective ASCII characters.

■ ANSI standard 6.8.6, The #pragma directive

<Behavior of #pragma preprocessing directives in the compiler>

Any #pragma preprocessing directive that cannot be interpreted is ignored.
When the compile option -Wunknown_pragma (-WUP) is specified, the #pragma directives that cannot be interpreted are warned.

■ ANSI standard 6.8.8, Predefined macro names

<Definitions of __DATE__ and __TIME__>

The macro names __DATE__ and __TIME__ are usable at all times.

I.2.14 Library Functions

■ ANSI standard 7.1.6, Common definition headers <stddef.h>

<Null pointers expanded by NULL>

The macro constant NULL (null pointer) is defined to be 0.

■ ANSI standard 7.2, Diagnostics headers <assert.h>

<Diagnostic messages by assert function>

The diagnostic messages output at termination of an assert function are as follows:
"Assertion failed: expression, file name (line number)"
If, while a macro NDEBUG is defined, a false expression is passed to an assert function, the program will loop infinitely in the abort library function without returning from the assert function.

■ ANSI standard 7.3.1, Character testing functions

<Character sets inspected by character testing functions>

The following lists the characters (ASCII characters) for which the functions isalnum, isalpha, iscntrl, islower, isupper, and isprint return true.

Function name	Character set
isalnum	0 to 9, A to Z, a to z
isalpha	A to Z, a to z
iscntrl	0x00 to 0x1F, 0x7F
islower	a to z
isupper	A to Z
isprint	0x20 to 0x7E

■ ANSI standard 7.5.1, Treatment of error conditions

<Values returned by mathematical functions when a domain error occurs>

If a domain error occurs in any mathematical function, macro EDOM is stored in errno.

<Whether mathematical functions will set errno to the value of macro ERANGE when an underflow error occurs>

If an underflow error occurs in any mathematical function, macro ERANGE is stored in errno.

■ ANSI standard 7.5.6.4, The fmod function

<When the second argument to the fmod function is 0>

If the second argument to the fmod function is 0, a domain error is assumed and macro EDOM is stored in errno. At this time, the value 0 is returned.

■ ANSI standard 7.7.1.1, The signal function

The C standard library of the M16C C compiler does not have the signal function implemented in it.

■ ANSI standard 7.9.2, Streams

<Whether a new-line character is needed for the last line of a text stream (text file)>

The standard library functions are designed to behave normally even if the last line is not accompanied by new-line code.

<Whether space characters written out to a text stream immediately before a new-line character will be output when read in>

The space characters preceding a new-line character also are output.

<Number of null characters added to a binary stream>

No null characters are appended to the end of a binary stream.

■ ANSI standard 7.9.3, Files

<Position of a file position indicator in an append-mode stream>

The streams of the M16C C compiler do not support the file position indicator.

<Whether a write to a text stream will cause the associated file to be truncated beyond that point>

In no event will a text stream be truncated.

<File buffering characteristics>

The M16C C compiler does not buffer libraries for input/output to and from files.

<Whether a file of zero length actually exists>

The M16C C compiler, in its libraries for input/output to and from files, does not exercise control on files to which actually output. The file system is outside the range of support provided by the M16C C compiler.

<Rules for making valid file names>

The M16C C compiler, in its libraries for input/output to and from files, does not exercise control on files to which actually output. The file system is outside the range of support provided by the M16C C compiler.

<Whether the same file can be simultaneously opened multiple times>

The M16C C compiler does not support the library functions associated with the operation to open or close files.

■ ANSI standard 7.9.4.1, The remove function

<Effect of the remove function in option files>

The C standard library of the M16C C compiler does not support file management.

The remove function is not implemented

■ ANSI standard 7.9.4.2, The rename function

<If a file with a new name exists before the rename function is called, what will happen to the file>

The C standard library of the M16C C compiler does not support file management.

The rename function is not implemented

■ ANSI standard 7.9.6.1, The fprintf function

<Output of %p conversion in the fprintf function>

As for output of the conversion specification %p of printf-series, the function outputs a 6-digit hexadecimal number as 24-bit address. A colon ": "is output between the 2 high-order digits and the 4 low-order digits.

■ ANSI standard 7.9.6.2, The fscanf function

<Input of %p conversion in the fscanf function>

As for input of the conversion specification %p of scanf-series, the function reads input of a hexadecimal number as 24-bit address. A colon ": " is required between the 2 high-order digits and the 4 low-order digits.

<Interpretation of a hyphen in scanf-series>

A hyphen in %[conversion is interpreted as an ordinary character.

■ ANSI standard 7.9.9.1, The fgetpos function, and 7.9.9.4, The ftell function

The C standard library of the M16C C compiler does not have the fgetpos and ftell functions implemented in it.

■ ANSI standard 7.9.10.4, The perror function

<Messages generated by the perror function>

One of the following strings is generated.

- String received as argument
- A domain error
- A range error

■ ANSI standard 7.10.3, Memory management functions

<Behavior of the calloc, malloc, and realloc functions when memory in size of 0 bytes is requested>

When a memory space in size of 0 bytes is requested for the calloc, malloc, or realloc functions, a NULL pointer is returned.

■ ANSI standard 7.10.4.1, The abort function

<Behavior of the abort function on open files and temporary files>

An infinite loop is entered into.

■ ANSI standard 7.10.4.3, The exit function

<Termination status returned by the exit function (if the argument value is none of 0, EXIT_SUCCESS, or EXIT_FAILURE)>

Nothing is returned.

■ ANSI standard 7.10.4.4, The getenv function

<How environment names are set and the environment list is altered in the getenv function>

The C standard library of the M16C C compiler does not have the getenv function implemented in it.

■ ANSI standard 7.10.4.5, The system function

<Content of string and mode of execution by the system function>

The C standard library of the M16C C compiler does not have the system function implemented in it.

■ ANSI standard 7.11.6.2, The strerror function

<Error messages returned to the strerror function>

The C standard library of the M16C C compiler is such that no functions are called in the strerror function.

■ ANSI standard 7.12.1, Components of time data

<Local time zone and daylight saving time>

Not supported for the M16C C compiler.

■ ANSI standard 7.12.2.1, The clock function

<Passage of time in the clock function>

The C standard library of the M16C C compiler does not have the clock function implemented in it.

I.3 Locale-Specific Behavior

With regard to the actions handled as "locale-specific behavior" in ANSI standards, the following describes behavior in C language under the M16C C compiler NC30.

The numbers and headings following "■ ANSI standard" denote the corresponding section numbers and section titles in ANSI standard "ANSI/ISO 9899—1990." The items handled as "locale-specific behavior" are enclosed in angle brackets < >, and the corresponding operation of NC30 are explained after the brackets.

■ ANSI standard 5.2.1, Character sets
<Content of the extended execution character set>
The characters defined in JISX0201 (not including Latin alphabets) and JISX0208 can be used.

■ ANSI standard 5.2.2, Character display semantics
<Direction of writing>
The direction of writing is from left to right.

■ ANSI standard 7.1.1, Definition of library terms
<Characters denoting a decimal point>
The character used to denote a decimal point, in all locales, is 0x2E ('.').

■ ANSI standard 7.3, Character handling function headers <ctype.h>
<Test character sets in the character testing functions>
For the implementation-defined set of characters to be tested by the character testing functions of character handling functions, please see ANSI standard 7.3.1, Character testing functions in 4.2.14, "Library Functions," of Section 4.2, "Implementation-Defined Behavior." In all locales, the macros and functions defined and declared in ctype.h behave the same way as in "C" environment ("C" locale).

■ ANSI standard 7.11.4.4, The strncmp function
<The order in which character sets are compared>
In all locales, the order in which character sets are compared in the strncmp function is the same as in ASCII.

■ ANSI standard 7.12.3.5, The strftime function
The C standard library of the M16C C compiler does not have the strftime function implemented in it.

Appendix J ELF Format Converter ELFCONV

This chapter describes how to start the ELF format converter ELFCONV, the functionality of its startup options, and the precautions to take when using it.

J.1 Overview

The ELF format converter ELFCONV is a utility tool that converts files in IEEE format to those in ELF format.
This utility tool permits the IEEE format object files or IEEE format libraries generated by NC30 V.6 or earlier version to be converted into the ELF format, so that they can be included in the projects built by NC30 V.6.00 or newer.

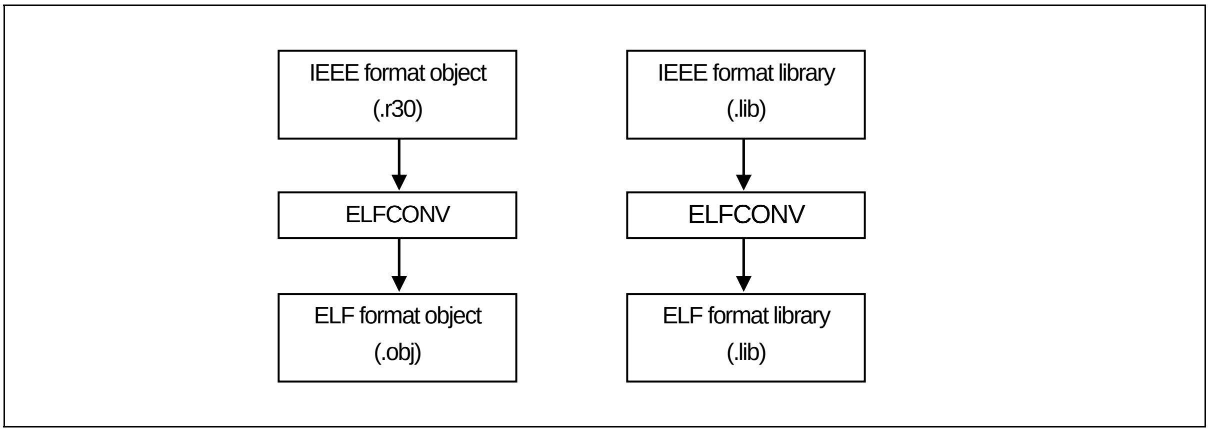


Figure J.1 Processing Flow of ELFCONV

J.2 Starting Up

J.2.1 Command Line Syntax

The following shows how to launch ELFCONV. For input, specify an object file name in IEEE format (extension .r30) or a library file name in IEEE format (extension .lib).

ELFCONV△[option]△IEEE format object filename (.r30)△-o△output filename△-cpu△CPU type<RET>
ELFCONV△[option]△IEEE format library filename (.lib)△-o△output filename△-cpu△CPU type<RET>

J.2.2 Options

The options specifiable for ELECONV are listed below.

Table J.1 List of ELFCONV Options

Option	Function
-o△output file name	Specifies an output file name. Specify different files for the input file and output file. If the input and output files have the same name, conversion is not performed and an error message is output.
-V	Only displays the version of ELFCONV and finishes. The versions of the internal tools are displayed at the same time.
-cpu△CPU type	Specifies a CPU type for the ELF file to be generated. Following can be specified for the CPU type: M16C(m16C), R8C(r8C), R8CE(R8ce) If the CPU type specified with this option and the CPU type of the input object file differ, the one specified with the option is assumed for the CPU type of the output object file. In this case, a warning is output.
-Wno_check_cpu	When this option is specified, a warning that would otherwise be output for the difference in CPU type is suppressed.

J.3 Precautions to Taken When Using ELFCONV

When using ELFCONV, pay attention to the following:

- The debug information included in the object or library of the input file is not subject to conversion.
- When you link the files converted by ELFCONV, do not use the -cpu=stride option of optlnk.
- Conversion by ELFCONV is performed in such a way that the result is the same as input 'in binary.' In the following cases, however, the result is not always the same:
  (1) Where sections that have the same name exist separately in multiple files
  (2) Where alignment specification in each section differs
  (3) Where the specified start position of a section (as in a link option) is an odd address
- The variable symbols specified by #pragma address are converted as absolute addresses. For this reason, variable information in the linker is subject to the following limitations:
  (1) In the .map files that are output when the list or show=reference,xreference option is specified, the number of times the variable symbols specified by #pragma address are referenced is 0 times.
  (2) A 'symbol-unreferenced' message, or the one that is output when the msg_unused option is specified, is output for the variable symbols specified by #pragma address.
- The objects converted by ELFCONV are excluded from the application of SBDAT/SPECIAL features by UTL30.
- If multiple instances of a section with the same name are defined, they are combined into one section. If there is a space between each section, it is filled with NOP code (0x04). Also, if multiple sections with the same name have different attributes, ELFCONV outputs an error or warning on display device according to the table below.

Table J.2 Sections with the same name have different attributes

Section address		Type of section		
Compare	Compared with	CODE	ROMDATA	DATA
Relative	Relative	Normal	Normal	Normal
Absolute	Relative	Normal	Normal	Normal
Absolute	Absolute (upper)	Error	Error	Warning
Absolute	Absolute (overlap)	Error	Error	Warning
Absolute	Absolute (lower)	Error	Error	Warning
Relative	Absolute	Error	Error	Error

J.4 ELFCONV Messages

The messages displayed by ELFCONV are listed below.

Table J.3 ELFCONV Message List

No.	Error or warning	Message	Solution
H1001	Warning	Address is overlapped in 'DATA' section 'section name'	Addresses in DATA section are overlapping.
H1002	Warning	Warning Absolute-section 'section name' is written after the same name of Absolute-section.	For a section specified as having absolute attribute, another section with the same name as that is specified as absolute.
H1005	Warning	Specified CPU type 'CPU type' is different from the object CPU type 'CPU type' in 'input module name'. 'CPU type' is adopted	The CPU type of the input object differs from the CPU type specified with an option.
H2001	Error	Cannot open file (file name).	The input file cannot be opened in read mode.
H2003	Error	File format error.	The input file is not created in IEEE695 format.
H2005	Error	Input file name is not specified.	No input files are specified.
H2010	Error	Unknown option (input option name)	Invalid option is input.
H2013	Error	Address is overlapped in 'CODE' section 'section name'	Addresses in CODE section are overlapping.
H2014	Error	Address is overlapped in 'ROMDATA' section 'section name'	Addresses in ROMDATA section are overlapping.
H2015	Error	Absolute-section 'section name' is written after the same name of Relocatable-section.	For a section specified as having relative attribute, another section with the same name as that is specified as absolute.

Appendix K Contents of Upgrade and Migration Method

K.1 Contents of Upgrade

This chapter describes the contents of upgrade from old versions, how to migrate the user application, and the precautions to take when migrating.

K.1.1 C++ Language Support

The files written in C++ (extensions .cpp, .cc, or .cp) can be compiled.

K.1.2 Conversion of the Integrated Development Environment (High-performance Embedded Workshop) Projects

The integrated development environment (High-performance Embedded Workshop) projects created by an old version of the compiler can be converted into projects for this version of the compiler.
When you're using the assembler startup, because -order is nonexistent, you need to set the following optlnk option of -start after porting projects to the current compiler version.

```
-start=data_SE,bss_SE,data_SO,bss_SO,data_NE,bss_NE,data_NO,bss_NO,stack,
istack,heap,rom_NE,rom_NO/0400,data_FE,bss_FE,data_FO,bss_FO/010000,
rom_FE,rom_FO,data_SEI,data_SOI,data_NEI,data_NOI,data_FEI,data_FOI,
switch_table,program,interrupt/0E0000,program_S/0F0000,vector/0FFD00
```

Also, since the start address of each section is assigned the default value, you need to change them to those that are set in your application.

K.1.3 Change of Object Formats

The object format has been changed from the conventional IEEE-695 to ELF/DWARF2 format. When you specify the object format to be loaded by the debugger, be sure to specify ELF/DWARF2.
A new tool, called the object converter (elfconv.exe), is provided for converting object files in IEEE-695 to the ELF format, so use it to convert your old files as necessary.

K.1.4 Change of File Extensions

The extension of object files has been changed from ".r30" to ".obj". Also, the extension of absolute files has been changed from ".x30" to ".abs".
If you're using the integrated development environment (High-performance Embedded Workshop), you do not specifically need to be concerned about it. The change is required when makefiles are used.

K.1.5 Change of Librarians

The librarian has been changed from the conventional "lb30" to "optlnk." When you use it on the command line, correct specification including the options, etc.

Function	lb30 option	Option in optlnk
Suppresses message output	-.	None
Adds module	-A file.lib file1.r30 file2.r30	-form=library -library=file.lib file1.obj file2.obj
Generates library file	-C file.lib file1.r30 file2.r30	-form=library -output=file.lib file1.obj file2.obj
Removes module	-D file.lib file1.r30	-form=library -library=file.lib -delete=file1
Creates library list file	-L file.lib -L file.lib file1.r30	-form=library -library=file.lib -list -show=symbol None
Replaces module	-R file.lib file1.r30	-form=library -library=file.lib -replace=file1.obj
Updates module	-U file.lib file1.r30	None
Displays version	-V	None
Extracts module	-X file.lib file1.r30	-library=file.lib -extract=file1 -form=object
Command file	@file	-subcommand=file.cmd

K.1.6 Change of Linkage Editors

The linkage editor has been changed from the conventional "ln30" to "optlnk." When you use it on the command line, correct specification including the options, etc.

Function	ln30 option	Option in optlnk
Suppresses message output	-.	None
Specifies entry point	-E point -E 0f0000	-entry=point -entry=0f0000
Outputs debug information	-G None -G	-debug -nodebug
Optimizes branch instruction	-JOPT	-optimize=branch
Specifies library	-l file.lib	-library=directory\file.lib
	-LD directory	
	Environment variable: LIB30	Environment variable: HLNK_DIR
Supports burn-into-ROM	-LOC PP=0fe000 -ORDER PP=00400	-rom=PP=PP_ram -start=PP_ram/00400,PP/fe000
Outputs linkage list	-M	-list
	-MS	-list -show=symbol
	-MSL	
Specifies upper-limit number of messages	-NOSTOP	None
Specifies output file name	-O file Extension is .x30.	-output=file Default extension when omitted is .abs.
Specifies location address	-ORDER AA=f0000,BB,CC	-start=AA,BB,CC/f0000
Outputs error tag file	-T	None
Notifies unnecessary symbol	-U	-msg_unused -message
Displays version	-V	None
Specifies vector	-VECT label -VECT 0ff000	-VECT=label -VECT=0ff000
	-VECTN label,20 -VECTN 0ff000,21	-VECTN=20=label -VECTN=21=0ff000
Suppresses load module generation when warned	-W	-change_message=error
Command file	@file	-subcommand=file.cmd
Specifies MCU	-M60 -M61 -R8C -R8CE	None

Set the environment variable HLNK_DIR for the path to the folder in which the library files for optlnk are searched, as necessary.

If the address value begins with a–f, ln30 options require a zero. Note, however, that optlnk options do not require a zero.

K.1.7 Change of Load Module Converters

The load module converter has been changed from the conventional "lmc30" to "optlnk." When you use it on the command line, correct specification including the options, etc.

Function	lmc30 option	Option in optlnk
Suppresses message output	-.	None
Specifies output address range	-A 1000:11ff -A 1000	-output=file.mot=1000-11ff -output=file.mot=1000-fffff
Specifies execution start address	-E 0f0000	-entry=0f0000
Sets data in free space	-F 00 -F 00:1000:10ff -F 00:1000	-space=00 None None
Generates file in Intel HEX format	-H	-form=hexadecimal
Sets ID code	-ID	Set by assembler directive .ID and ID files output by optlnk
Selects data length	-L	-byte_count=20
Specifies output file name	-O file	-output=file
Specifies OFSREG set value	-ofsregx -protect1 -protectx	Set by assembler directive .OFSREG Set by assembler directive .PROTECT
Displays version	-V	None
Generates file in Motorola S format	Default	-form=stype
Controls generated code	-R8C -R8CE	None

K.1.8 Change of Stack Amount Usage Calculation Utilities

The utilities that calculate the stack amounts used have been changed from the conventional "stk.exe," "stkviewer.exe," and "CallWalker.exe" to only "CallWalker.exe." When sources are linked after adding the -stack option in optlnk, stack files (.sni) are generated.

K.1.9 Change of Map Information Display Tools

The tool to display map information has been changed from the conventional Map Viewer.exe to only the map window of the integrated development environment (High-performance Embedded Workshop).

K.1.10 Use of a Library Generator

A library generator (lbg30.exe) has been introduced that generates a standard library according to user-specified options. If you're using the integrated development environment (High-performance Embedded Workshop), you do not specifically need to be concerned about it. To use on the command line, however, invoke the library generator directly or use the standard library files underneath the LIB30 folder (nc30lib.lib, nc30s16.lib, r8celib.lib, r8ces16.lib, r8clib.lib, or r8cs16.lib).

K.1.11 Change of Display Messages

The formats in which error and warning messages are output by the compiler and assembler have been changed.This makes it possible to use the error message jump feature of the integrated development environment (High-performance Embedded Workshop).

K.1.12 Addition of Compile Options

The following compile options have been added:

(1) -preinclude, which specifies include files
(2) -lang, which specifies the language to compile
(3) -exception and -noexception, which turn the exception handling facility of C++ on or off
(4) -rtti, which is the runtime type identification facility of C++
(5) -lnkcmd, which specifies the options to be passed to the linker
(6) -goptimize, which outputs additional information for intermodule optimization

K.1.13 Addition of Assembler Directives

The following assembler directives have been added:

(1) .reserve_area, which reserves storage for an area
(2) .words, which stores signed 2-bytes long data
(3) .callind, which defines inspector information on function calls

K.1.14 Addition of Assembler Options

The following assembler options have been added:

(1) -goptimize, which outputs additional information for intermodule optimization
(2) -subcommand, which passes options to the assembler via a file

K.1.15 Change of Methods for Setting External Jump Optimization

In old versions, we used the compile option -OGJ, assembler option -JOPT, and link option -JOPT to optimize external jumps. In this version, use the compile option -goptimize and assembler option -goptimize.

K.1.16 Disused Facilities

Assembler options	-M60 -M61 -M62E -JOPT	Disused.
Compiler options	-gbool_to_char -gold -Oglb_jmp -Werror_file -Wmake_tagfile -Wstdout -ln30 -fJSRW	Disused.
Assembler directives	.ver .optj .sjmp	Disused.
Pragma directives	#pragma JSRA	If -Wunknown_pragma(-WUP) or -Wall is used, a warning is generated for a description of #pragma JSRA, so delete the description concerned.
	#pragma JSRW	If -Wunknown_pragma(-WUP) or -Wall is used, a warning is generated for a description of #pragma JSRA, so delete the description concerned.
	#pragma STRUCT xxx arrange #pragma PAGE	Unusable in C++. In C, however, they can be used the same way as in the past.
	#pragma ROM	Disused.
Tools	Absolute lister (abs30.exe)	In the debugger's disassemble window, use Save File.
	Cross referencer (xrf30.exe)	Use map files generated by optlnk.
Files	SPECIAL page vector definition file special.inc	Disused.

K.2 Precautions to Take when Migrating

K.2.1 About Linking of Objects Generated by -R8C Option

The objects generated by the -R8C option can only be linked themselves together, and cannot be linked with those otherwise generated. So please be aware of it.

K.2.2 About Linking of Objects Generated by -R8CE Option

The objects generated by the -R8CE option can only be linked themselves together, and cannot be linked with those otherwise generated. So please be aware of it. Also, when you use the -R8CE option, link the generated file with one of the following libraries:

(1) Standard libraries generated using the -R8CE option for the library generator
(2) r8celib.lib (However, only when you've used nc30lib.lib previously)
(3) r8ces16.lib (However, this applies only when nc30s16.lib has been used before)

K.2.3 About Specification Change when Symbols with the Same Name Exist in Multiple Library Files

In the conventional ln30, if symbols with the same name existed in multiple library files, the symbol in the object file that was first input when creating library files was enabled.
In optlnk, if symbols with the same name exist, there is provided a facility that generates a warning. Also, optlnk has a facility that lets you specify the order of link one library module at a time and one that lets you change global symbols to the local attribute. Using these facilities, you now can link your intended symbols forcibly.

K.2.4 About Handling of .section Description in #pragma ASM/ENDASM

Do not write the assembler directive .section in the #pragma ASM/ENDASM areas within a function. The compiler does not detect it, so please be aware of it. If outside a function, you can write it, though.

K.2.5 About Warning Display during Intermodule Optimization

If, while using the assembly language startup (ncrt0.a30), you use intermodule optimization in a ported project, the following warning may be output.

** L1001 (W) Option "optimize=symbol_delete" is ineffective without option "entry"

In this case, specify -entry=start for optlnk. Here, "start" denotes the symbol that is set in the reset vector.

K.2.6 When Using Only the Standard Library Functions sprintf, vsprintf, and sscanf

When a program that uses only the standard C library functions sprintf, vsprintf, and sscanf is linked, the __sget, __iob, $_fp, or $_sput symbols may result in an undefined error. In that case, create a dummy stub function described in Section 2.2.2, "Customizing the Assembler Startup Program," before linking.

K.2.7 Altering the Assembler Startup

Alter the assembler startup as described below.

(1) C++ library initialization/termination process invocation

When you use C++, invoke C++ library initialization and termination processes before and after a call to _main in ncrt0.a30, as shown below.

```
                .glb        __CALL_INIT
                .call       __CALL_INIT,G
                jsr.a       __CALL_INIT

                .glb        _main
                .call       _main,G
                jsr.a       _main

                .glb        _exit
                .glb        $exit
                .glb        __exit_loop
_exit:
$exit:
                .glb        __CALL_END
                .call       __CALL_END,G
                jsr.a       __CALL_END
```

(2) Alteration of a description of section initialization

If you continue using the assembler startup of an old version, be sure to change the section initialization process in the ncrt0.a30 file to a description like the one shown below that uses "top of xxxx" to indicate the beginning of each section.

```
;---------------------------------------------------------------
; bss zero clear
;---------------------------------------------------------------
                N_BZERO     (topof bss_SE),bss_SE
                N_BZERO     (topof bss_SO),bss_SO
                N_BZERO     (topof bss_NE),bss_NE
                N_BZERO     (topof bss_NO),bss_NO

;---------------------------------------------------------------
; initialize data section
;---------------------------------------------------------------
                N_BCOPY     (topof data_SEI),(topof data_SE),data_SE
                N_BCOPY     (topof data_SOI),(topof data_SO),data_SO
                N_BCOPY     (topof data_NEI),(topof data_NE),data_NE
                N_BCOPY     (topof data_NOI),(topof data_NO),data_NO

;====================================================================
; FAR area initialize.
;---------------------------------------------------------------
; bss zero clear
;---------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                BZERO       (topof bss_FE),bss_FE
                BZERO       (topof bss_FO),bss_FO
.endif

;---------------------------------------------------------------
; initialize data section
;---------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                BCOPY       (topof data_FEI),(topof data_FE),data_FE
                BCOPY       (topof data_FOI),(topof data_FO),data_FO
.if __STACKSIZE__ != 0
```

(3) Alteration of a SECT30.INC description

If you continue using the assembler startup files of an old version, note that an optlnk startup warning message "L1322(W) Section alignment mismatch" may be output. If this warning is output, delete ALIGN descriptions in data_NEI and data_FEI section definitions (.section) from the sect30.inc file.

Also, if you specify -fno_align(-fNA) or -OR_MAX(-ORM), delete the following description from the sect30.in file.

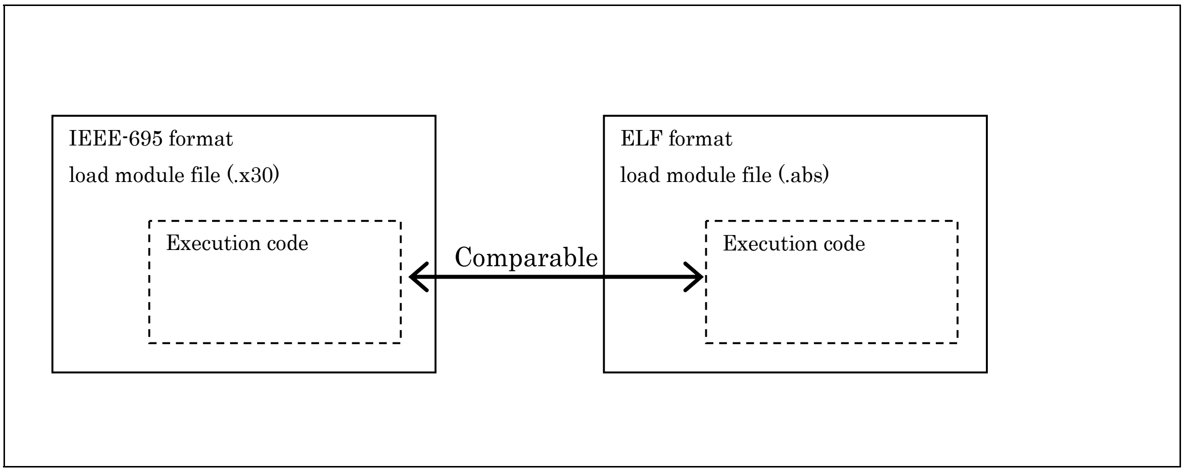
```
        .section      program,CODE,ALIGN
        .section      program_S,CODE,ALIGN
```

K.3 About Execution Code Comparison/Verification after Object Format Conversion

The object format conversion tool (elfconv.exe) is a tool that lets you convert the files in IEEE-695 format generated by old versions, i.e., the objects (.r30) [hereafter referred to as IEEE-r30] and libraries (.lib) [hereafter referred to as IEEE-lib], to the ELF format files used in NC V.6.00, i.e., the objects (.obj) [hereafter referred to as ELF-obj] and libraries (.lib) [hereafter referred to as ELF-lib].
The tool elfconv.exe only converts file formats and does not affect execution code. Therefore, you can use execution code that has a proven track record in NC30 by converting its format.
The following describes a method for verifying that execution code is unaffected by the object format conversion tool.

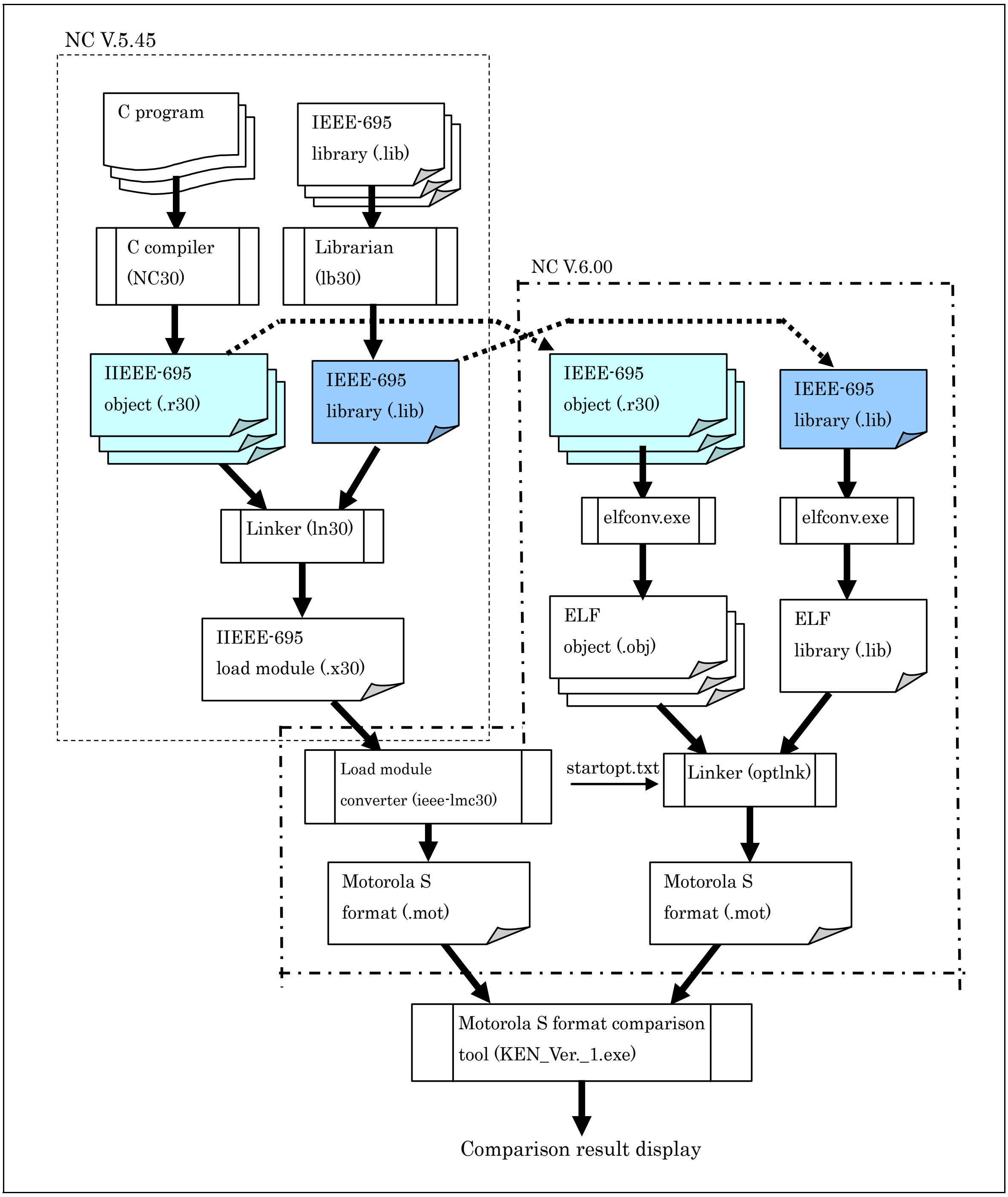
K.3.1 Concept of the Verification Method

Since IEEE-r30 and ELF-obj differ in format, they cannot be compared for verification directly as they are.
Therefore, generate machine-language files from the above files before performing comparison.



K.3.2 Procedure of the Verification Method

A concrete procedure of the verification method is illustrated below.



Step 1. Preparation

Prepare the IEEE-r30 or IEEE-lib that you used in an old version and the load module file in IEEE-695 format (.x30) obtained after linking [hereafter referred to as IEEE-x30]. When you are using multiple libraries, extract all object files using the librarian (lb30) and combine them into one library.

Also, be sure that the environment variables needed for NC V.6.00 (BIN30, LIB30, INC30, TMP30, and PATH) are correctly set.

Example:

sample1.r30	(object file in IEEE-695 format)
sample2.r30	(object file in IEEE-695 format)
sample3.lib	(library file in IEEE-695 format)
sample.x30	(load module file in IEEE-695 format)

Step 2. Generation of MOT files of NC V.5.45

Using ieee-lmc30.exe included with the NC V.6.00 package, generate MOT files in Motorola S format and startopt.txt (section mapping information) from IEEE-x30.

Note, however, that the -optlnk option must always be set when you invoke ieee-lmc30.exe. Unless this option is set, startopt.txt is not generated.

Example:

```
$ %BIN30%\ieee-lmc30.exe -optlnk -o sample_ieee sample.x30
```

(sample_ieee.mot and startopt.txt are produced.)

Step 3. Execution of object conversion

Using elfconv.exe, convert object formats from IEEE-r30 or IEEE-lib to ELF-obj or ELF-lib. Be aware that both IEEE-695 and ELF libraries have the same file extension ".lib".

See Appendix J for the -cpu option to elfconv.exe.

Example:

```
$ "%BIN30%\elfconv.exe "-o sample1.obj sample1.r30 -cpu m16c
$ "%BIN30%\elfconv.exe "-o sample2.obj sample2.r30 -cpu m16c
$ "%BIN30%\elfconv.exe "-o sample3_elf.lib sample3.lib -cpu m16c
```

(sample1.obj, sample2.obj, and sample3_elf.lib are produced.)

Step 4. Generation of load module files in ELF format

Using the linkage editor optlnk.exe, link the ELF-obj or ELF-lib files you've created in step 3 to generate MOT files in Motorola S format. For the -start option of optlnk.exe, specify the startopt.txt generated in step 3.

Example:

```
$ %BIN30%\optlnk.exe -form=stype -output=sample_elf.mot
               -list=sample_elf.map -sub=startopt.txt
               sample1.obj sample2.obj -library=sample3_elf.lib
```

(sample_elf.mot is produced.)

Step 5. Comparison of MOT files

<Only for PCs with Japanese-version operating systems>

Compare the IEEE-MOT files generated in step 2 with the ELF-MOT files generated in step 4 by using the Gen 1 S-format comparison tool, which is available from Renesas. For details on how to obtain and use Gen 1, see K.3.4, About the Gen 1 S-Format File Comparison Tool.

For comparison on PCs with English-version operating systems, refer to the release notes.

K.3.3 Precautions

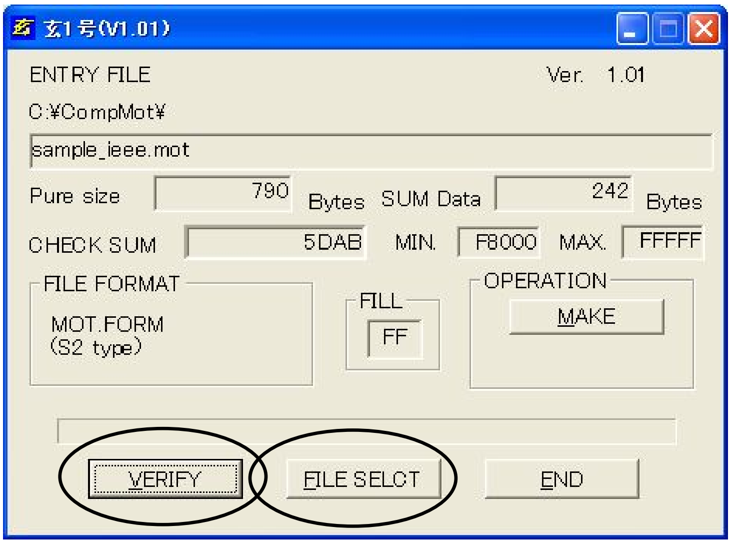
- If there are duplicate module names in one library and another, pick out the necessary modules and link the necessary side of the duplicates.
- To compare for verification a group of files that cause an undefined symbol error at link time, create a stub, etc. and resolve the undefined symbols before performing comparison for verification.
- If the order of execution code in the files obtained after linking differs, those files cannot be compared. Therefore, adjust the -start option at link time to ensure that the order of execution code in the generated files is the same. Referring to map files will prove convenient.
- If the ID information, protect information, or optional facility select register is not set, difference between the generated MOT files will result. In that case, use the assembler directives .id, .protect, or .ofsreg to set the necessary information.

K.3.4 About S Format File Comparison Tool "Ken 1"

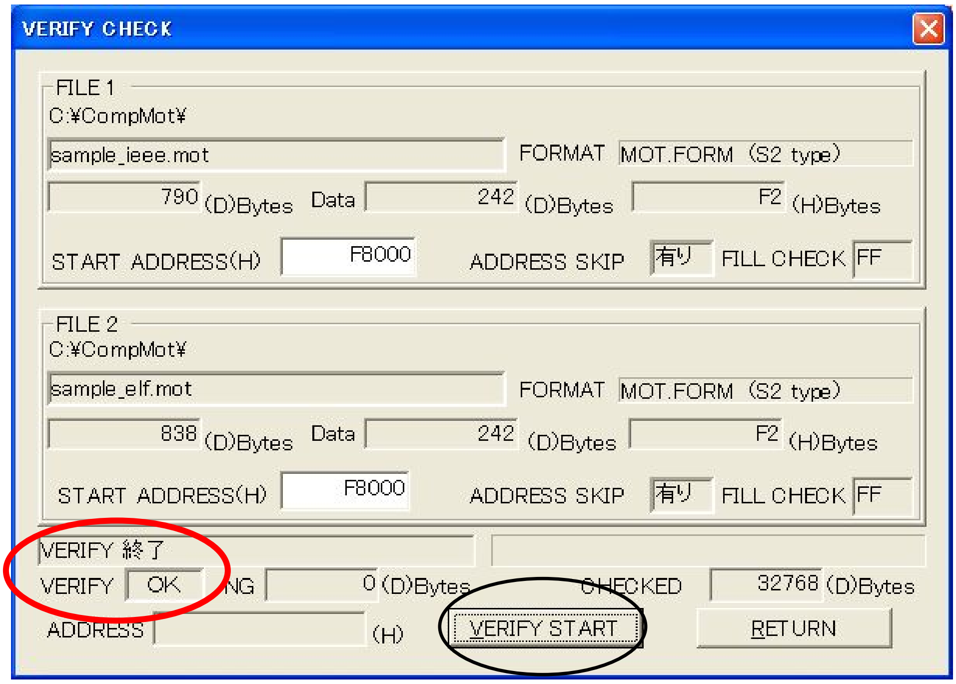
The S format file comparison tool is a program that calculates checksum values, which you would use at the time you order ROMs from Renesas.

1. Where to obtain from
   Download the latest version from the URL given below and extract it to an appropriate folder.
   http://japan.renesas.com/products/mpumcu/rom_ordering/superh_rom_ordering/child_folder/checksum_child.jsp

2. Double-click on extracted KEN_Ver._1.exe to launch, and a dialog like the one shown below will be displayed. Using the FILE SELECT button in the dialog, specify one MOT file to compare. Next, use the VERIFY button to specify the other MOT file.



3. The display will change to a dialog like the one shown below. Click VERIFY START to start a comparison.



4. If you see "VERIFY FINISHED" or "VERIFY OK" displayed at the lower corner of the dialog, it means that the comparison is finished, with the result that there is no difference between the MOT files.

Appendix L　Precautions

There are following precautions to taken when using this product.

L.1　Precautions Concerning the MCU-Dependent Part

L.1.1　Precautions Concerning Access to the SFR Area

To access registers in the SFR area, it will sometime be necessary to use a specific instruction.
Since this specific instruction differs with each MCU type, refer to the user's manual for the MCU type used.
For the instructions associated with this precaution, use the inline assemble facility of an asm function, etc. to write such an instruction directly in the program.

L.1.2　Regarding the M16C/62 4M Extension Mode

Be sure that the program is mapped to the internal ROM.

L.1.3　Regarding the FirmRam_NE Section and SB Register Value when On-Chip Debugger is Selected

If any debugger is selected in the OnChipDebugger select window when creating a new project workspace, the FirmRam_NE section may be mapped to memory beginning with 400H. Since the initial value of the SB register is set with 400H, the program will become unable to access the correct area in SB relative addressing mode.
If, as a result of linking, the FirmRam_NE section is found to have been mapped to memory beginning with 400H, change the initial value of the SB register to the start address of the bss_SE section. Check the map file to confirm the start address of the bss_SE section.
Change the values at two spots below to the start address of the bss_SE section.

<resetprg.c>

```
void start(void)
{
        :
        _sb_        = 0x400;   // 400H fixation (Do not change)

}
```

<resetprg.h>

```
#define   DEF_SBREGISTER          _asm("         .glb          __SB__  ¥n"¥
                                       "__SB__   .equ          0400H")
```

The MCUs concerned are as follows (as of May 16, 2009):
M16C/26, M16C/26A, M16C/28, M16C/29,
M16C/30P,
M16C/62P,
M16C/6N4, M16C/6N5, M16C/6NK, M16C/6NL, M16C/6NM, M16C/6NN,
M16C/6S,
M16C/64,
M16C/64A,
M16C/65

L.2 Precautions Concerning the Compiler, Assembler, Linkage Editor, and Utility

L.2.1 About -ffar_pointer(-fFP)

If, while -ffar_pointer is specified, you use the & operator to obtain the address of a variable that has the near attribute, the pointer is handled in 16 bits. So cast with a far pointer before the & operator.
Also, if a pointer size is obtained with sizeof, the return value becomes 2. When a function is called that has no prototype declaration, only 2 bytes are stacked for the address. Always be sure to declare prototype for the function to be called.

L.2.2 About the Standard I/O Functions

The standard input/output functions such as printf, etc. consume a large amount of RAM. Therefore, when you use the standard input/output functions in a library for the R8C family MCU (-R8C option specified), be aware that the conversion specifying symbols %e, %E, %f, %g, and %G cannot be used. This restriction applies to r8clib.lib, r8cs16.lib, and the libraries created using the -nofloat option of the library generator lbg30.exe.

L.2.3 Precautions Concerning the Inline Assemble Facility (#pragma ASM to #pragma ENDASM, asm Function)

- About program descriptions in #pragma ASM to #pragma ENDASM and asm function

(1) The compiler analyzes the program flow with respect to the live range of registers and that of variables before processing. Therefore, do not write in an asm function, etc. a branch (including a conditional branch) that may affect the program flow.

(2) The compiler analyzes the scope of the arguments passed via register and that of register variables as it generates code. However, if a description is included in the program that manipulates register variables using the inline assemble facility (#pragma ASM to #pragma ENDASM or an asm function), the C compiler is unable to hold information on scopes of these arguments and register variables that take effect in a program part in which the inline assemble facility is written. Therefore, when you write a process to manipulate registers using the inline assemble facility, be sure to save and restore the registers.

L.2.4 Precautions Concerning the Memory Management Functions malloc(), calloc(), and realloc()

The memory management functions "malloc()," "calloc()," and "realloc()" of NC30WA cannot reserve storage of more than 64 KB at a time.

L.3 Regarding Conformance to MISRA C Rules

L.3.1 Standard Function Library

Although the C source code of the M3T-NC30WA standard function library is found to contain a few violations of MISRA C rules[1], there are no hindrances to behavior attributable to those violations.

L.3.2 Causes of Violations of Rules

Listed below are the primary causes of violations of rules found in the C source code of the M3T-NC30WA standard function library:

- C compiler specifications (near/far qualifiers, asm() function, and #pragma)
- Function declarations based on ANSI standard
- Absence of an explicit description by parentheses ( ) of the order of evaluation in conditional statements
- Implicit type conversion

L.3.3 Inspection Numbers that Resulted in a Violation of Rules

The inspection numbers that resulted in a violation of rules are as follows:

1	12	13	14	18	21	22	28	34	35
36	37	38	39	43	44	45	46	48	49
50	54	55	56	57	58	59	60	61	62
65	69	70	71	72	76	77	82	83	85
99	101	103	104	105	110	111	115	118	119
121	124	-	-	-	-	-	-	-	-

L.3.4 Evaluation Environment

Compiler	M3T-NC30WA V.5.30 Release 1
Compile options	-O -c -as30 "-DOPTI=0" -gnone -finfo -fNII -misra_all -r $*.csv
MISRA C checker	SQMlint V.1.00 Release 1A

L.3.5 Source Code Automatically Generated by the Integrated Development Environment (High-performance Embedded Workshop)

Although the source code automatically generated by the integrated development environment (High-performance Embedded Workshop) is observed to contain a few violations of MISRA C rules, there are no hindrances to behavior attributable to those violations.

L.3.6 Causes of Violations of Rules

Listed below are the primary causes of violations of rules found in the source code generated by the integrated development environment (High-performance Embedded Workshop).

- C compiler specifications (#pragma, etc.)
- Scopes of variables defined in headers
- Definitions of types used in bit-fields

[1] This is the result of inspection performed by the MISRA C rule checker SQMLint.

L.3.7 Inspection Numbers that Resulted in a Violation of Rules

The inspection numbers that resulted in a violation of rules are as follows:

13	14	22	34	36	37	43	45	46
49	54	59	69	76	82	85	99	104
110	111	115	124	126	-	-	-	-

L.3.8 Evaluation Environment

Compiler	M3T-NC30WA V.5.45 Release00
Compile options	-c -misra_all
MISRA C checker	SQMlint V.1.03 Release 00

L.3.9 #pragma Extended Facilities Used in C Startup (Misra C Rule 99)

Extended facility	Declaration file	Content	Description
#pragma STACKSIZE	resetprg.h	Defines user stack size.	Stack section (stack) is output and top label name of the stack is generated.
#pragma ISTACKSIZE	resetprg.h	Defines interrupt stack size.	Interrupt stack section (istack) is output and top label name of the interrupt stack is generated.
#pragma CREG	resetprg.h	Declares MCU's internal registers.	When accessing the internal register declared by this pragma, code to access it using a dedicated instruction is generated.
#pragma sectaddress	resetprg.h fvector.c	Defines section. Location address can be declared at the same time.	Section is defined with the section name declared by this pragma. If address is specified at the same time, address definition using the directive .org is output.
#pragma entry	resetprg.h	Declares function to be executed at reset.	No enter instruction is output that builds stack frame for the function declared by this pragma. This is intended to prevent the enter instruction from being generated before stack pointer is initialized.
#pragma interrupt/V	fvetor.c	Generates vector table.	Only interrupt vector is defined for the function declared by this pragma.
#pragma inline	resetprg.h	Declares inline function.	The function declared by this pragma is expanded in-line.
#pragma interrupt	intprg.c fvector.c	Declares interrupt function.	For the function declared using this pragma, interrupt function code is generated.
#pragma section	heap.c resetprg.c initsct.h resetprg.c firm.c	Changes section name.	Section name is changed to that defined by this pragma.
#pragma ADDRESS	Each srf header file	Defines I/O address and declares variables.	For sfr defined by this pragma, address is defined with .equ.

C/C++ Compiler Package for M16C Series and R8C Family V.6.00
C/C++ Compiler User's Manual

Publication Date: Jan. 16, 2011 Rev.1.00

Published by: Renesas Electronics Corporation

Edited by: Renesas Solutions Corp.

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
7F, No. 363 Fu Shing North Road Taipei, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

© 2010 Renesas Electronics Corporation and Renesas Solutions Corp. All rights reserved.
Colophon 1.0

C/C++ Compiler Package
for M16C Series and R8C Family V.6.00
C/C++ Compiler User's Manual

Renesas Electronics Corporation

REJ10J2188-0100