User's Manual

NOTICE:
There are corrections in TableC.3 NC30 Specifications on page 175.
There are additions in -fbit(-FB) on page 88 and -Wlarge_to_small(-WLTS) on page 99.

M16C Series, R8C Family
C Compiler Package V.5.45
C Compiler User's Manual

Renesas Electronics
www.renesas.com

Rev.3.00 2010.11

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 assembly language 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. HP-UX is a registered trademark of Hewlett-Packard Company.
- IBM and AT are registered trademarks of International Business Machines Corporation.
- Intel and Pentium are registered trademarks of Intel Corporation.
- 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
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.

Chapter 1 Introduction to NC30 ... 1
1.1 NC30 Components ... 1
1.2 NC30 Processing Flow ... 2
1.2.1 NC30 ... 3
1.2.2 cpp30 ... 3
1.2.3 ccom30 ... 3
1.2.4 aopt30 ... 3
1.2.5 as30 ... 3
1.2.6 sbauto ... 3
1.2.7 ln30 ... 3
1.2.8 utl30 ... 3
1.2.9 genmap ... 3
1.2.10 gensni ... 4
1.3 Notes ... 5
1.3.1 Notes about Version-up of compiler ... 5
1.3.2 Notes about the M16C's Type Dependent Part ... 5
1.4 Example Program Development ... 6
1.5 NC30 Output Files ... 8
1.5.1 Introduction to Output Files ... 8
1.5.2 Preprocessed C Source Files ... 9
1.5.3 Assembly Language Source Files ... 11
Chapter 2 Basic Method for Using the Compiler ... 14
2.1 Starting Up the Compiler ... 14
2.1.1 nc30 Command Format ... 14
2.1.2 Command File ... 15
2.1.3 Notes on NC30 Command Line Options ... 16
2.1.4 nc30 Command Line Options ... 17
2.2 Preparing the Startup Program ... 23
2.2.1 Sample of Startup Program ... 23
2.2.2 Customizing the Startup Program ... 28
2.2.3 Customizing for NC30 Memory Mapping ... 32
Chapter 3 Programming Technique ... 44
3.1 Notes ... 44
3.1.1 Notes about Version-up of compiler ... 44
3.1.2 Notes about the M16C's Type Dependent Part ... 44
3.1.3 About Optimization ... 45
3.1.4 Precautions on Using register Variables ... 48
3.2 For Greater Code Efficiency ... 49
3.2.1 Programming Techniques for Greater Code Efficiency ... 49
3.2.2 Speeding Up Startup Processing ... 52
3.3 Linking Assembly Language Programs with C Programs ... 53
3.3.1 Calling Assembler Functions from C Programs ... 53
3.3.2 Writing Assembler Functions ... 56
3.3.3 Notes on Coding Assembler Functions ... 60
3.4 Other ... 61
3.4.1 Precautions on Transporting between NC-Series Compilers ... 61
Appendix A Command Option Reference ... 61
A.1 nc30 Command Format ... 61
A.2 nc30 Command Line Options ... 62
A.2.1 Options for Controlling Compile Driver ... 62
A.2.2 Options Specifying Output Files ... 65

A.2.3 Version Information Display Option 66
A.2.4 Options for Debugging 67
A.2.5 Optimization Options 69
A.2.6 Generated Code Modification Options 82
A.2.7 Library Specifying Option 95
A.2.8 Warning Options 96
A.2.9 Assemble and Link Options 103
A.3 Notes on Command Line Options 104
A.3.1 Coding Command Line Options 104
A.3.2 Priority of Options for Controlling 104
Appendix B Extended Functions Reference 106
B.1 Near and far Modifiers 108
B.1.1 Overview of near and far Modifiers 108
B.1.2 Format of Variable Declaration 108
B.1.3 Format of Pointer type Variable 109
B.1.4 Format of Function Declaration 111
B.1.5 near and far Control by nc30 Command Line Options 111
B.1.6 Function of Type conversion from near to far 112
B.1.7 Checking Function for Assigning far Pointer to near Pointer 112
B.1.8 Declaring functions 113
B.1.9 Function for Specifying near and far in Multiple Declarations 113
B.1.10 Notes on near and far Attributes 114
B.2 asm Function 115
B.2.1 Overview of asm Function 115
B.2.2 Specifying FB Offset Value of auto Variable 116
B.2.3 Specifying Register Name of register Variable 120
B.2.4 Specifying Symbol Name of extern and static Variable 121
B.2.5 Specification Not Dependent on Storage Class 124
B.2.6 Selectively suppressing optimization 125
B.2.7 Notes on the asm Function 125
B.3 Description of Japanese Characters 128
B.3.1 Overview of Japanese Characters 128
B.3.2 Settings Required for Using Japanese Characters 128
B.3.3 Japanese Characters in Character Strings 129
B.3.4 sing Japanese Characters as Character Constants 130
B.4 Default Argument Declaration of Function 131
B.4.1 Overview of Default Argument Declaration of Function 131
B.4.2 Format of Default Argument Declaration of Function 131
B.4.3 Restrictions on Default Argument Declaration of Function 133
B.5 inline Function Declaration 134
B.5.1 Overview of inline Storage Class 134
B.5.2 Declaration Format of inline Storage Class 134
B.5.3 Restrictions on inline Storage Class 135
B.6 Extension of Comments 138
B.6.1 Overview of "//" Comments 138
B.6.2 Comment "//" Format 138
B.6.3 Priority of "//" and "/*" 138
B.7 #pragma Extended Functions 139
B.7.1 Index of #pragma Extended Functions 139
B.7.2 Using Memory Mapping Extended Functions 143
B.7.3 Using Extended Functions for Target Devices 152

B.7.4 The Other Extensions 161
B.8 assembler Macro Function 166
B.8.1 Outline of Assembler Macro Function 166
B.8.2 Description Example of Assembler Macro Function 166
B.8.3 Commands that Can be Written by Assembler Macro Function 167
Appendix C Overview of C Language Specifications 174
C.1 Performance Specifications 174
C.1.1 Overview of Standard Specifications 174
C.1.2 Introduction to NC30 Performance 174
C.2 Standard Language Specifications 177
C.2.1 Syntax 177
C.2.2 Type 180
C.2.3 Expressions 182
C.2.4 Declaration 183
C.2.5 Statement 186
C.3 Preprocess Commands 188
C.3.1 List of Preprocess Commands Available 188
C.3.2 Preprocess Commands Reference 188
C.3.3 Predefined Macros 197
C.3.4 Usage of predefined Macros 197
Appendix D C Language Specification Rules 198
D.1 Internal Representation of Data 198
D.1.1 Integral Type 198
D.1.2 Floating Type 199
D.1.3 Enumerator Type 200
D.1.4 Pointer Type 200
D.1.5 Array Types 200
D.1.6 Structure types 201
D.1.7 Unions 202
D.1.8 Bitfield Types 202
D.2 Sign Extension Rules 204
D.3 Function Call Rules 205
D.3.1 Rules of Return Value 205
D.3.2 Rules on Argument Transfer 205
D.3.3 Rules for Converting Functions into Assembly Language Symbols 206
D.3.4 Interface between Functions 211
D.4 Securing auto Variable Area 217
D.5 Rules of Escaping of the Register 218
Appendix E Standard Library 219
E.1 Standard Header Files 219
E.1.1 Contents of Standard Header Files 219
E.1.2 Standard Header Files Reference 220
E.2 Standard Function Reference 228
E.2.1 Overview of Standard Library 228
E.2.2 List of Standard Library Functions by Function 229
E.2.3 Standard Function Reference 236
E.2.4 Using the Standard Library 305
E.3 Modifying Standard Library 306
E.3.1 Structure of I/O Functions 306
E.3.2 Sequence of Modifying I/O Functions 307
Appendix F Error Messages 314

F.1 Message Format ... 314
F.2 nc30 Error Messages ... 315
F.3 cpp30 Error Messages ... 317
F.4 cpp30 Warning Messages ... 320
F.5 ccom30 Error Messages ... 321
F.6 c ccom30 Warning Messages ... 334
Appendix F Error Messages ... 314
F.1 Message Format ... 314
F.2 nc30 Error Messages ... 315
F.3 cpp30 Error Messages ... 317
F.4 cpp30 Warning Messages ... 320
F.5 ccom30 Error Messages ... 321
F.6 c ccom30 Warning Messages ... 334
Appendix H Using gensni or the stack information File Creation Tool for Call Walker ... 357
H.1 Starting Call Walker ... 357
H.2 Outline of gensni ... 357
H.2.1 Processing Outline of gensni ... 357
H.3 Starting gensni ... 358
H.3.1 Input format ... 358
H.3.2 Option References ... 360

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 consists of the following executable files:

(1) nc30................................ Compile driver
(2) cpp30............................. Preprocessor
(3) ccom30........................... Compiler
(4) aopt30............................ Assembler Optimizer
(5) as30……………………Assembler System
(6) sbauto .... SB register automatic changeover utility
(7) ln30……………………Linkage Editor
(8) utl30............................... SBDATA declaration & SPECIAL page Function declaration Utility
(9) genmap.......................... Map-information generating tool for the High-performance Embedded Workshop
(10) gensni……………….. Stack calculation utility for the Call Walker

1.2 NC30 Processing Flow

Figure 1.1 illustrates the NC30 processing flow.

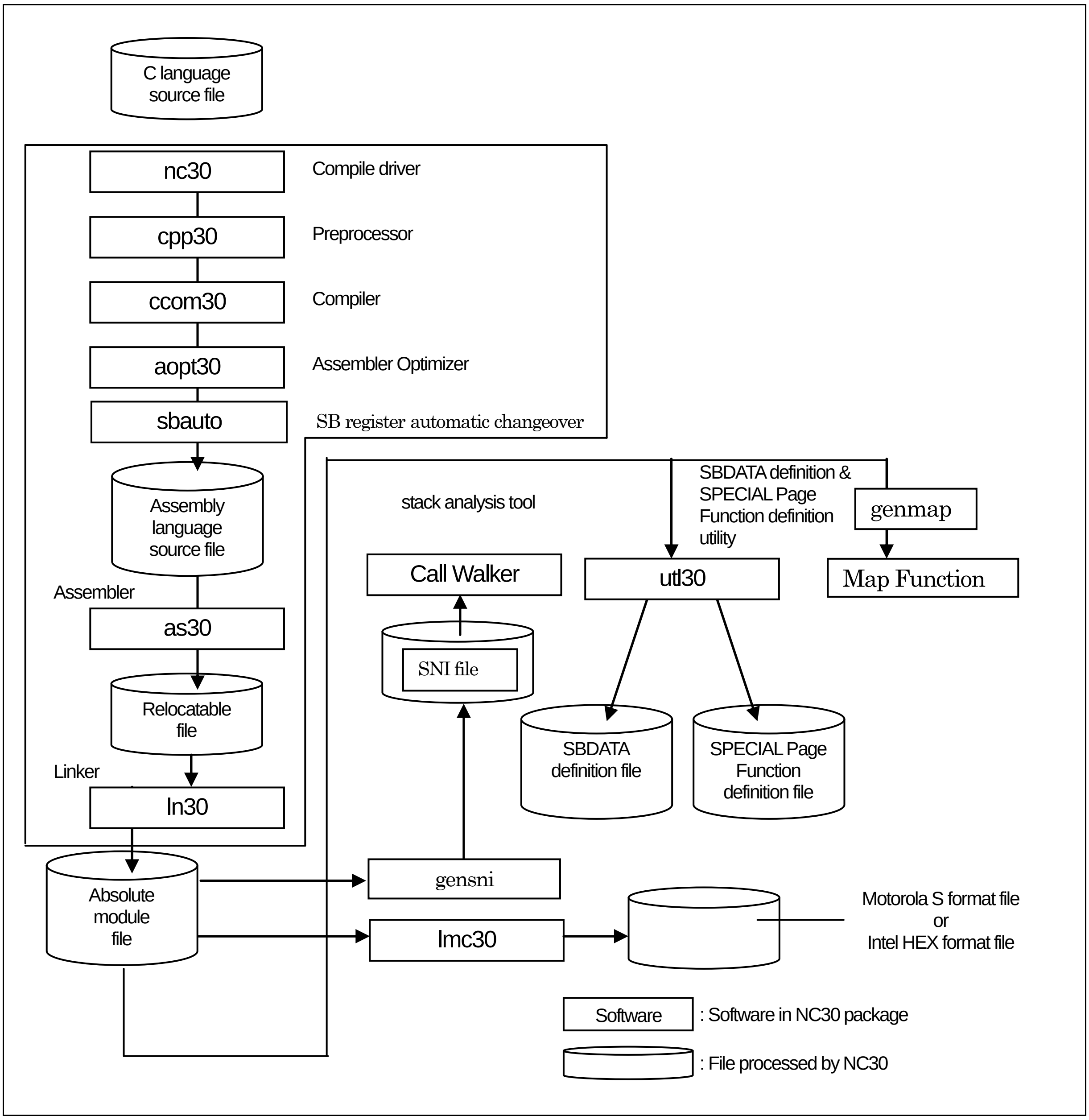


Figure 1.1 NC30 Processing Flow

1.2.1 NC30

NC30 is the executable file of the compile driver.
By specifying options, NC30 can perform the series of operations from compiling to linking. You can also specify for the as30 relocatable macro assembler and four for the ln30 linkage editor by including the -as30 and -ln30 command line options when you start NC30.

1.2.2 cpp30

cpp30 is the executable file for the preprocessor.
cpp30 processes macros starting with # (#define, #include, etc.) and performs conditional compiling (#if-#else-#endif, etc.).

1.2.3 ccom30

ccom30 is the executable file of the compiler itself.
C source programs processed by cpp30 are converted to assembly language source programs that can be processed by AS30.

1.2.4 aopt30

aopt30 is the assembler optimizer.
It optimizes the assembler codes output by ccom30.

1.2.5 as30

as30 is an assembler system, which assembles the assembler code output by ccom30.

1.2.6 sbauto

sbauto analyzes the number of times external variables are referenced in a function based on the inspector information that was output by the compiler, and outputs optimum SB relative.

1.2.7 ln30

ln30 is a linkage editor, which links multiple relocatable files and library files to generate absolute-module files (.x30).

1.2.8 utl30

utl30 is the execution file for the SBDATA declaration utility and SPECIAL page Function declaration Utility.
By processing the absolute module file (.x30), utl30 generates a file that contains SBDATA declarations (located in the SB area beginning with the most frequently used one) and a file that contains SPECIAL page function declarations (located in the SPECIAL page area beginning with the most frequently used one).
To use utl30, specify the compile driver startup option -finfo when compiling, so that the absolute module file (.x30) will be generated.

1.2.9 genmap

genmap is a utility that allows the use of the mapping facility of the High-performance Embedded Workshop. genmap is automatically executed when the mapping facility is activated.
The mapping facility of the High-performance Embedded Workshop loads the .map file output by genmap and graphically shows the memory map after linkage.
If you wish to use the mapping facility of the High-performance Embedded Workshop, specify "-finfo" (which is an option to start up the compile driver) at the time of compilation and generate an absolute-module file

(.x30).

1.2.10 gensni

gensni is a utility that analyzes the information required for the Call Walker.
The Call Walker loads the stack information file (.sni) output by gensni and indicates the stack size.
If you wish to use the Call Walker and gensni, specify "-finfo" (which is an option to start up the compile driver) at the time of compilation so that inspector information will be appended to the absolute-module file (.x30).

1.3 Notes

Renesas Electronics Corp. 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 Corp., 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.

1.3.1 Notes about Version-up of compiler

The machine-language instructions (assembly language) generated by NC30 vary in contents depending on the startup options specified when compiling, contents of version-up, etc. Therefore, when you have changed the startup options or upgraded the compiler version, be sure to reevaluate the operation of your application program.

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.

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.
In this compiler, the instructions which cannot be used may be generated for writing and read-out to the register of SFR area. When you describe like the following examples as C language description to a SFR area, in this compiler may generate the assembler code which carries out operation which is not assumed since the interrupt request bit is not normal.
When accessing registers in the SFR area in 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 */
}
```

Figure 1.2 C language description to SFR area

1.4 Example Program Development

Figure 1.3 shows the flow for the example program development using NC30. The program is described below.

(Items [1] to [4] correspond to the same numbers in Figure 1.3)

(1) The C source program AA.c is compiled using NC30, then assembled using as30 to create the re-locatable object file AA.r30.

(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) The modified startup program is assembled to create the relocatable object file ncrt0.a30.

(4) The two relocatable object files AA.r30 and ncrt0.a30 are linked by the linkage editor ln30, which is run from nc30, to create the absolute module file AA.x30.

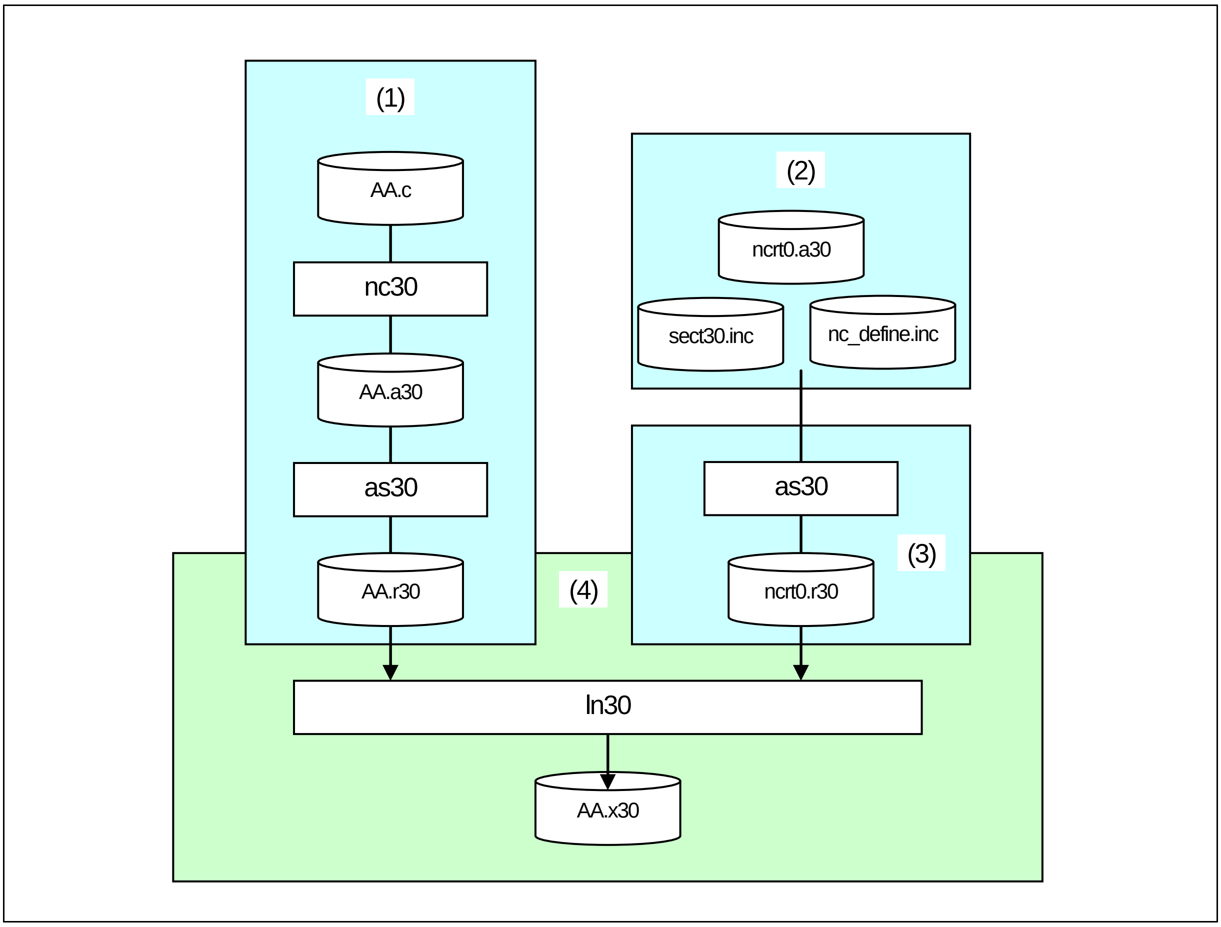


Figure 1.3 Program Development Flow

Figure 1.3 is an example make file containing the series of operations shown in Figure 1.4.

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

ncrt0.r30 : ncrt0.a30
          as30 ncrt0.a30

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

Figure 1.4 Example make File

Figure 1.5 shows the command line required for NC30 to perform the same operations as in the make file shown in Figure 1.4.

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

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

*Specify ncrt0.a30 first ,when linking.
```

Figure 1.5 Example NC30 Command Line

1.5 NC30 Output Files

This chapter introduces the preprocess result C source program output when the sample program sample.c is compiled using NC30 and the assembly language source program.

1.5.1 Introduction to Output Files

With the specified command line options, the NC30 compile driver outputs the files shown in Figure 1.6. Below, we show the contents of the files output when the C source file smp.c shown in Figure 1.7 is compiled, assembled, and linked.

See the AS30 User Manual for the relocatable object files (extension .r30), print files (extension .lst),and map files (extension .map) output by as30 and ln30.

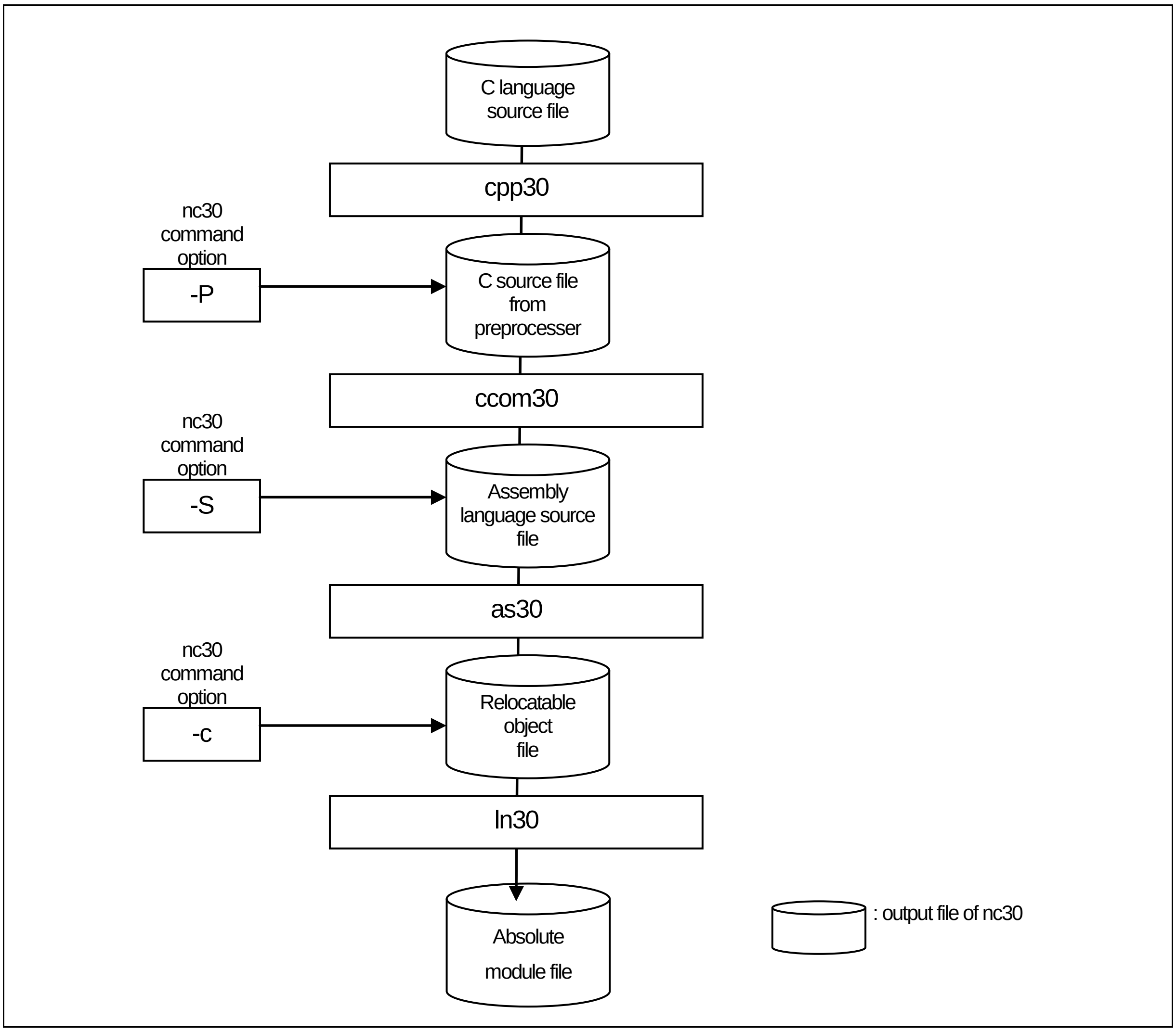


Figure 1.6 Relationship of NC30 Command Line Options and Output Files

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

void        main(void)
{
            int         flag;

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

Figure 1.7 Example C Source File (sample.c)

1.5.2 Preprocessed C Source Files

The cpp30 processes preprocess commands starting with #. Such operations include header file contents, macro expansion, and judgements on conditional compiling.

The C source files output by the preprocessor include the results of cpp30 processing of the C source files. Therefore, do not contain preprocess lines other than #pragma and #line. You can refer to these files to check the contents of programs processed by the compiler. The file extension is .i.

Figure 1.8 and Figure 1.9 are examples of file output.

```
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[];
```

Figure 1.8 Example Preprocessed C Source 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    init_prn(void);
extern int    _sget(void);
extern int    _sput(int);
extern int    _pput(int);
extern const char _far *_print(int(*)(), const char _far *, int _far * _far *, int _far *);
```

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

              flag = 0 ;                             ← (3)

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

}
```

Figure 1.9 Example Preprocessed C Source File (2)

Let's look at the contents of the preprocessed C source file.
Items (1) to (4) correspond to (1) to (4) in Figure 1.8 and Figure 1.9.

(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, because PRN specified in #define is 1, the compile condition is satisfied and the printf function is output.

1.5.3 Assembly Language Source Files

The assembly language source file is a file that can be processed by AS30 as a result of the compiler ccom30 converting the preprocess result C source file. The output files are assembly language source files with the extension .a30.

Figure 1.10 and Figure 1.11 are examples of the output files. When the NC30 command line option "-dsource (-dS) " is specified, the assembly language source files contain the contents of the C source file as comments.

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

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

;## COMMAND_LINE: ccom30   C:¥Renesas¥nc30wa¥v544r00¥TMP¥sample.i -o .¥sample.a30 -dS

;## Normal Optimize              OFF                                  (1)
;## ROM size Optimize            OFF
;## Speed Optimize               OFF
;## Default ROM is               far
;## Default RAM is               near
                .GLB        __SB__
                .SB         __SB__
                .FB         0

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


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

Figure 1.10 Example Assembly Language Source File (1) "sample.a30"

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

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

Figure 1.11 Example Assembly Language Source File (2) "sample.a30"

Let's look at the contents of the assembly language source files. Items (1) to (2) correspond to (1) to (2) in Figure 1.10.

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

(2) When the NC30 command line option "-dsource (-dS)" is specified, shows the contents of the C source file(s) as commen

Chapter 2 Basic Method for Using the Compiler

This chapter describes how to start the compile driver nc30 and the command line options.

2.1 Starting Up the Compiler

2.1.1 nc30 Command Format

The nc30 compile driver starts the compiler commands (cpp30 and ccom30), the assemble command as30 and the link command ln30 to create a absolute module file. The following information (input parameters) is needed in order to start nc30:

(1) C source file(s)
(2) Assembly language source file(s)
(3) Relocatable object file(s)
(4) Command line options (optional)

These items are specified on the command line.

Figure 2.1 shows the command line format. Figure 2.2 is an example. In the example, the following is performed:

(1) Startup program ncrt0.a30 is assembled.
(2) C source program sample.c is compiled and assembled.
(3) Relocatable object files ncrt0.r30 and sample.r30 are linked.

The absolute module file sample.x30 is also created. The following command line options are used:

- Specifies machine language data file sample.x30...................... option -o
- Specifies output of list file (extension .lst) at assembling........... option -as30 "-l"
- Specifies output of map file (extension .map) at linking........... option -ln30 "-ms"

```
% nc30∆[command-line-option]∆[assembly-language-source-file-name]∆
                                [relocatable-object-file-name]∆<C-source-file-name>

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

Figure 2.1 nc30 Command Line Format

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

<RET> : Return key
* Always specify the startup program first when linking.
```

Figure 2.2 Example nc30 Command Line

2.1.2 Command File

The compile driver can compile a file which has multiple command options written in it (i.e., a command file) after loading it into the machine.
Use of a command file helps to overcome the limitations on the number of command line characters imposed by PC, etc.

a. Command file input format

```
% nc30Δ[command-line-option]Δ<@file-name>[command-line-option]

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

Figure 2.3 Command File Command Line Format

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

<RET> : Return key
* Always specify the startup program first when linking.
```

Figure 2.4 Example Command File Command Line

Command files are written in the manner described below.

```
Command File description              ncrt0.a30<CR>
                                      sample1.c   sample2.r30<CR>
                          ------->    -g   -as30   -l<CR>
                                      -o<CR>
<CR>: Denotes carriage return.        sample<CR>
```

Figure 2.5 Example 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 file 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.
Command files for ln30 whose file name extension is ".cm$" are automatically generated in order for specifying files when linking. Therefore, existing files with the file name extension ".cm$", if any, will be overwritten. Do not use files which bear the file name extension ".cm$" along with this compiler. You cannot specify two or more command files simultaneously.
If multiple files are specified, the compiler displays an error message "Too many command files".

2.1.3 Notes on NC30 Command Line Options

a. Notes on Coding nc30 Command Line Options

The nc30 command line options differ according to whether they are written in uppercase or lowercase letters. Some options will not work if they are specified in the wrong case.

b. Priority of Options for Controlling Compile driver

Priority of Options for Controlling Compile driver.

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

Therefore, if the following two options are specified at the same time, for example,

- "-c": Finish processing after creating a relocatable module file (extension .r30)
- "-S": Finish processing after creating an assembly language source file (extension .a30) the -S option has priority. That is to say, the compile driver does not perform any further processing after assembling.

In this case, it only generates an assembly language source file. If you want to create a re-locatable file simultaneously with an assembly language source file, use the option "-dsource(shortcut -dS)".

2.1.4 nc30 Command Line Options

a. Options for Controlling Compile Driver

Table 2.1 shows the command line options for controlling the compile driver. The details of each optional notes please refer to Appendix A.

Table 2.1 Options for Controlling Compile Driver

Option	Function
-c	Creates a relocatable module file (extension .r30) and ends processing.[1]
- D*identifier*	Defines an identifier. Same function as #define.
-dsource (Short form -dS)	Generates an assembly language source file (extension ".a30") with a C language source list output as a comment. (Not deleted even after assembling.)
-dsource_in_list (Short form -dSL)	In addition to the "-dsource" function, generates an assembly language list file (.lst).
-E	Invokes only preprocess commands and outputs result to standard output.
-I*directory*	Specifies the directory containing the file(s) specified in #include. You can specify up to 256 directories.
-P	nvokes only preprocess commands and creates a file (extension .i).
-S	Creates an assembly language source file (extension .a30) and ends processing.
-silent	Suppresses the copyright message display at startup.
-U*predefined macro*	Undefines the specified predefined macro.

b. Options Specifying Output Files

Table 2.2 shows the command line option that specifies the name of the output machine language data file.

Table 2.2 Options for Specifying Output Files

Option	Function
-dir*directory-name*	Specifies the destination directory of the file(s) (absolute module file, map file, etc.) generated by ln30.
-o*file-name*	Specifies the name(s) of the file(s) (absolute module file, map file, etc.) generated by ln30. This option can also be used to specify the destination directory. Do not specify the filename extension.

[1] If you do not specify command line options -c, -E, -P, or -S, nc30 finishes at ln30 and output files up to the absolute load module file (extension .x30) are created.

c. Version and command line Information Display Option

Table 2.3 shows the command line options that display the cross-tool version data and the command line informations.

Table 2.3 Options for Displaying Version Data and Command line informations

Option	Function
-v	Displays the name of the command program and the command line during execution.
-V	Displays the startup messages of the compiler programs, then finishes processing .(without compiling)

d. Options for Debugging

Table 2.4 shows the command line options for outputting the symbol file for the C source file.

Table 2.4 Options for Debugging

Option	Function
-g	Outputs debugging information to an assembler source file (extension .a30).Therefore you can perform C language- level debugging.
-genter	Always outputs an enter instruction when calling a function. Be sure to specify this option when using the debugger's stack trace function.
-gno_reg	Suppresses the output of debugging information for register variables.
-gbool_to_char	This option outputs bool-type debugging information as the char type.
-gold	This option outputs debugging information in Rev.E format. When this option specifies, the “-gno_reg” option and the “-fauto_128” option are automatically specified.

e. Optimization Options

Table 2.5 shows the command line options for optimizing program execution speed and ROM capacity.

Table 2.5 Optimization Options

Option	Short form	Function
-O[1-5]	None	Optimization of speed and ROM size.
-OR	None	Optimization of ROM size.
-OS	None	Optimization of speed.
-OR_MAX	-ORM	Places priority on ROM size for the optimization performed.
-OS_MAX	-OSM	Places priority on speed for the optimization performed.
-Ocompare_byte_to_word	-OCBTW	Compares consecutive bytes of data at contiguous addresses in words.
-Oconst	-OC	Performs optimization by replacing references to the const-qualified external variables with constants.
-Oforward_function_to_inline	-OFFTI	Expands all inline functions in-line.
-Oglb_jmp	-OGJ	Global jump is optimized.
-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	Inhibits starting 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 data.
-Ono_float_const_fold	-ONFCF	Suppresses the constant folding processing of floating point numbers.
-Ono_logical_or_combine	-ONLOC	Suppresses the optimization that puts consecutive OR 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 allows the necessary ROM capacity to be reduced. However, this may result in an increased amount of stack being used.
-Ostack_frame_align	-OSFA	Aligns the stack frame on an every boundary.
-Ostatic_to_inline	-OSTI	A static function is treated as an inline function.
-O5OA	None	Inhibits code generation based on bit-manipulating instructions when the optimization option “-O5” is selected.

f. Generated Code Modification Options

Table 2.6 to Table 2.7 shows the command line options for controlling nc30-generated assembly code.

Table 2.6 Generated Code Modification Options (1)

Option	Short form	Function
-fansi	None	Makes "-fnot_reserve_far_and_near", "-fnot_reserve_asm", and "-fextend_to_int" valid.
-fchar_enumerator	-fCE	Handles the enumerator type as an unsigned char type, not as an int type.
-fconst_not_ROM	-fCNR	Does not handle the types specified by const as ROM data.
-fdouble_32	-fD32	This option specifies that the double type be handled in 32-bit data length as is the float type.
-fenable_register	-fER	Make register storage class available.
-fextend_to_int	-fETI	Performs operation after extending char-type data to the int type. (Extended according to ANSI standards.) [2]
-ffar_RAM	-fFRAM	Changes the default attribute of RAM data to far.
-finfo	None	Outputs the information required for the "Call Walker", "Map Function", and "utl30" to the absolute module file (.x30).
-fJSRW	None	Changes the default instruction for calling functions to JSR.W.
-fbit	-fB	Generates code assuming that bitwise manipulating instructions can be executed using absolute addressing for all external variables mapped into the near area.
-fno_carry	-fNC	Suppresses carry flag addition when data is indirectly accessed using far-type pointers.
-fauto_128	-fA1	Limits the usable stack frame to 128 bytes.
-ffar_pointer	-fFP	Change the default attribute of pointer-type variable to far.
-fnear_ROM	-fNROM	Change the default attribute of ROM data to near.
-fno_align	-fNA	Does not align the start address of the function.
-fno_even	-fNE	Allocate all data to the odd section, with no separating odd data from even data when outputting .
-fno_switch_table	-fNST	When this option is specified, the code which branches since it compares is generated to a switch statement.
-fnot_address_volatile	-fNAV	Does not regard the variables specified by #pragma ADDRESS (#pragma EQU) as those specified by volatile.
-fnot_reserve_asm	-fNRA	Exclude asm from reserved words. (Only _asm is valid.)
-fnot_reserve_far_and_near	-fNRFAN	Exclude far and near from reserved words. (Only _far and _near are valid.)
-fnot_reserve_inline	-fNRI	Exclude far and near from reserved words. (Only _inline is made a reserved word.)
-fsmall_array	-fSA	When referencing a far-type array whose total size is unknown when compiling, this option calculates subscripts in 16 bits assuming that the array's total size is within 64 Kbytes.
-fswitch_other_section	-fSOS	This option outputs a ROM table for a 'switch' statement to some other section than a program section.
-fchange_bank_always	-fCBA	This option allows you to write multiple variables to an extended area.

[2] char-type data or signed char-type data evaluated under ANSI rules is always extended to inttype data.
This is because operations on char types (c1=c2*2/c3; for example) would otherwise result in an overflow and failure to obtain the intended result.

Table 2.7 Generated Code Modification Options (2)

Option	Short form	Function
-fauto_over_255	-fAO2	Changes the stack frame size per function that can be reserved to 64K bytes.
-fsizet_16	-fS16	Change the type definition size_t from type unsigned long to type unsigned int
-fptrdifft_16	-fP16	Change the type definition ptrdiff_t from type signed long to type signed int
-fuse_DIV	-fUD	This option changes generated code for divide operation.
-fuse_MUL	-fUM	This option changes generated code multiple operation.
-R8C	None	Generates object code for R8C Family.
-R8CE	None	Generates code suitable for the R8C Family with 64-KB or larger ROM.

g. Library Specifying Option

Table 2.8 lists the startup options you can use to specify a library file.

Table 2.8 Library Specifying Option

Option	Function
-l*libraryfilename*	Specifies a library file that is used by ln30 when linking files.

h. Warning Options

Table 2.9 shows the command line options for outputting warning messages for contraventions of nc30 language specifications.

Table 2.9 Warning Options

Option	Short form	Function
-Wall	None	Displays message for all detectable warnings. (however, not including alarms output by -Wlarge_to_small and "-Wno_used_argument")
-Wccom_max_warnings =*Warning Count*	-WCMW	This option allows you to specify an upper limit for the number of warnings output by ccom30.
-Werror_file<*file name*>	-WEF	Outputs error messages to the specified file.
-Wlarge_to_small	-WLTS	Outputs a warning about the tacit transfer of variables in descending sequence of size.
-Wmake_tagfile	-WMT	Outputs error messages to the tag file of source file by source file.
-Wnesting_comment	-WNC	Outputs a warning for a comment including "*/" .
-Wno_stop	-WNS	Prevents the compiler stopping when an error occurs.
-Wno_used_argument	-WNUA	Outputs a warning for unused argument of functions.
-Wno_used_function	-WNUF	Displays unused global functions when linking.
-Wno_used_static_function	-WNUSF	For one of the following reasons, a static function name is output that does not require code generation.
-Wno_warning_stdlib	-WNWS	Specifying this option while "-Wnon_prototype" or "-Wall" is specified inhibits "Alarm for standard libraries which do not have prototype declaration.
-Wnon_prototype	-WNP	Outputs warning messages for functions without prototype declarations.
-Wstdout	None	Outputs error messages to the host machine's standard output (stdout).
-Wstop_at_link	-WSAL	Stops linking the source files if a warning occurs during linking to suppress generation of absolute module files. Also, a return value "10" is returned to the host OS.
-Wstop_at_warning	-WSAW	Stops compiling the source files if a warning occurs during compiling and returns the compiler end code "10".
-Wundefined_macro	-WUM	Warns you that undefined macros are used in #if.
-Wuninitialize_variable	-WUV	Outputs a warning about auto variables that have not been initialized.
-Wunknown_pragma	-WUP	Outputs warning messages for non-supported #pragma.

i. Assemble and Link Options

Table 2.10 shows the command line options for specifying as30 and ln30 options.

Table 2.10 Assemble and Link Options

Option	Function
-as30Δ< Option>	Specifies options for the as30 link command. If you specify two or more options, enclose them in double quotes.
-ln30Δ< Option>	Specifies options for the ln30 assemble command. If you specify two or more options, enclose them in double quotes.

2.2 Preparing the Startup Program

For C-language programs to be "burned" into ROM, NC30 comes with a sample startup program written in the assembly language to initial set the hardware (M16C/60), locate sections, and set up interrupt vector address tables, etc. This startup program needs to be modified to suit the system in which it will be installed. The following explains about the startup program and describes how to customize it.

2.2.1 Sample of Startup Program

The NC30 startup program consists of the following two files:

- ncrt0.a30
  Write a program which 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 vector and special-page vector.
- sect30.inc
  Included from ncrt0.a30, this file defines section locations (memory mapping).

Figure 2.6 to Figure 2.10 show the ncrt0.a30 and nc_define.inc source program lists.

```
[ncrt0.a30]
;====================================================================
; heap area initialize                                                  ←(1)
;---------------------------------------------------------------------
.if __HEAPSIZE__ != 0
                .glb            __mnext
                .glb            __msize
                mov.w           #(heap_top&0FFFFH),__mnext
                mov.w           #(heap_top>>16),__mnext+2
                mov.w           #(__HEAPSIZE__&0FFFFH),__msize
                mov.w           #(__HEAPSIZE__>>16),__msize+2
.endif

(1) initializes the heap area.
[nc_define.inc]

__HEAPSIZE__    .equ    0300H   ; HEEP SIZE definition                  ←(2)

(2) defines the heap size.
```

Figure 2.6 Startup Program List (1) (ncrt0.a30 and nc_define.inc)

[nc_define.inc]

```
__STACKSIZE__       .equ     0300H     ; STACK SIZE definition                        ←(3)
__ISTACKSIZE__      .equ     0300H     ; INTERRUPT STACK SIZE definition              ←(4)

__VECTOR_ADR__       .equ    0ffd00H   ; INTERRUPT VECTOR ADDRESS definition          ←(5)
__SPECIAL_PRG__     .equ 0f8000H       ; Special page program address                 ←(6)
```

[ncrt0.a30]

```
;---------------------------------------------------------------------
; include files
;---------------------------------------------------------------------
                    .list               OFF
                    .include   nc_define.inc                                          ←(7)
                    .include   sect30.inc                                             ←(8)
                    .list               ON
```

(3) defines the user stack size.
(4) defines the interrupt stack size.
(5) defines the start address of interrupt vector table.
(6) defines the start address of the special-page vector table.
(7) Includes nc_define.inc
(8) Includes sect30.inc

Figure 2.7 Startup Program List (2) (ncrt0.a30,nc_define.inc)

```
;====================================================================
; Interrupt section start
;--------------------------------------------------------------------
        .insf       start,S,0
        .glb        start
        .section    interrupt
start:                                                        ← (9)
;--------------------------------------------------------------------
; after reset,this program will start
;--------------------------------------------------------------------
        ldc         #istack_top,isp      ;set istack pointer
        mov.b       #02h,0ah
        mov.b       #00h,04h             ;set processer mode   ← (10)
        mov.b       #00h,0ah
.if __STACKSIZE__ != 0
        ldc         #0080h,flg                                ← (11)
        ldc         #stack_top,sp        ;set stack pointer
.else
        ldc         #0000h,flg
.endif
        ldc         #data_SE_top,sb      ;set sb register
        ldintb      #__VECTOR_ADR__
;====================================================================
; NEAR area initialize.
;--------------------------------------------------------------------
; bss zero clear                                              ← (12)
;--------------------------------------------------------------------
        N_BZERO bss_SE_top,bss_SE
        N_BZERO bss_SO_top,bss_SO
        N_BZERO bss_NE_top,bss_NE
        N_BZERO bss_NO_top,bss_NO

;--------------------------------------------------------------------
; initialize data section                                     ← (13)
;--------------------------------------------------------------------
        N_BCOPY data_SEI_top,data_SE_top,data_SE
        N_BCOPY data_SOI_top,data_SO_top,data_SO
        N_BCOPY data_NEI_top,data_NE_top,data_NE
        N_BCOPY data_NOI_top,data_NO_top,data_NO
```

(9) After a reset, execution starts from this label (start)
(10) Sets processor operating mode
(11) Switches between the user stack and interrupt stack.
(12) Clears the near bss section (to zeros).
(13) Transfers the initial value of the data section in the near area to the RAM.

Figure 2.8 Startup Program List (3) (ncrt0.a30)

```
;=====================================================================
; FAR area initialize.
;---------------------------------------------------------------------
; bss zero clear                                                        ←(14)
;---------------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                BZERO           bss_FE_top,bss_FE
                BZERO           bss_FO_top,bss_FO
.endif

;---------------------------------------------------------------------
; initialize data section                                               ←(15)
;---------------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                BCOPY           data_FEI_top,data_FE_top,data_FE
                BCOPY           data_FOI_top,data_FO_top,data_FO
.if __STACKSIZE__ != 0
                ldc                     #stack_top,sp
.else
                ldc                     #istack_top,isp
.endif
                .stk            -40
.endif
;=====================================================================
; heap area initialize                                                  ←(16)
;---------------------------------------------------------------------
.if __HEAPSIZE__ != 0
                .glb            __mnext
                .glb            __msize
                mov.w           #(heap_top&0FFFFH),__mnext
                mov.w           #(heap_top>>16),__mnext+2
                mov.w           #(__HEAPSIZE__&0FFFFH),__msize
                mov.w           #(__HEAPSIZE__>>16),__msize+2
.endif
;=====================================================================
; Initialize standard I/O                                               ←(17)
;---------------------------------------------------------------------
.if __STANDARD_IO__ == 1
                .glb            __init
                .call           __init,G
                jsr.a           __init
.endif
;=====================================================================
; Call main() function                                                  ←(18)
;---------------------------------------------------------------------
                ldc             #0h,fb                  ; for debuger

                .glb            _main
                jsr.a           _main
```

(14) Clears the far bss section (to zeros).
(15) Moves the initial values of the far data section to RAM.
(16)) Initializes the heap area. Comment out this line if no memory management function is used.
(17) Calls the init function, which initializes standard I/O. Comment out this line if no I/O function is used.
(18) Calls the 'main' function.
* Interrupt is not enable, when calls 'main' function. Therefore, permits interrupt by FSET command, when uses interrupt function.

Figure 2.9 Startup Program List (4) (ncrt0.a30)

```
;=======================================================================
; exit() function                                                       ← (19)
;-------------------------------------------------------------------
        .glb    _exit
        .glb    $exit
_exit:                          ; End program
$exit:
        jmp     _exit
        .einsf

;=======================================================================
; dummy interrupt function                                              ← (20)
;-------------------------------------------------------------------
        .glb    dummy_int
dummy_int:
        reit
        .end
;
(19) exit function.
(20) Dummy interrupt processing function.
```

Figure 2.10 Startup Program List (5) (ncrt0.a30)

2.2.2 Customizing the Startup Program

a. Overview of Startup Program Processing

(1) About ncrt0.a30

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

- Sets the top address (__SB__) of the SBDATA area (it is accessing area to used the SB relative addressing mode).
- Sets the processor's operating mode.
- Initializes the stack pointer (ISP Register and USP Register).
- Initializes SB register.
- Initializes INTB register.
- Initializes the data near area.

  bss_NE bss_NO bss_SE and bss_SO sections are cleared (to 0).

  Also, the initial values in the ROM area (data_NEI, data_NOI, data_SEI, data_SOI) are transferred to RAM (data_NE ,data_NO, data_SE and data_SO).
- Initializes the data far area.

  bss_FE and bss_FO sections are cleared (to 0).

  Also, the initial values in the ROM area (data_FEI, data_FOI) storing them are transferred to RAM (data_FE, data_FO).
- Initializes the heap area.
- Initializes the standard I/O function library.
- Initializes FB register .
- Calls the 'main' function.

b. Modifying the Startup Program

Figure 2.11 summarizes the steps required to modify the startup programs to match the target system.

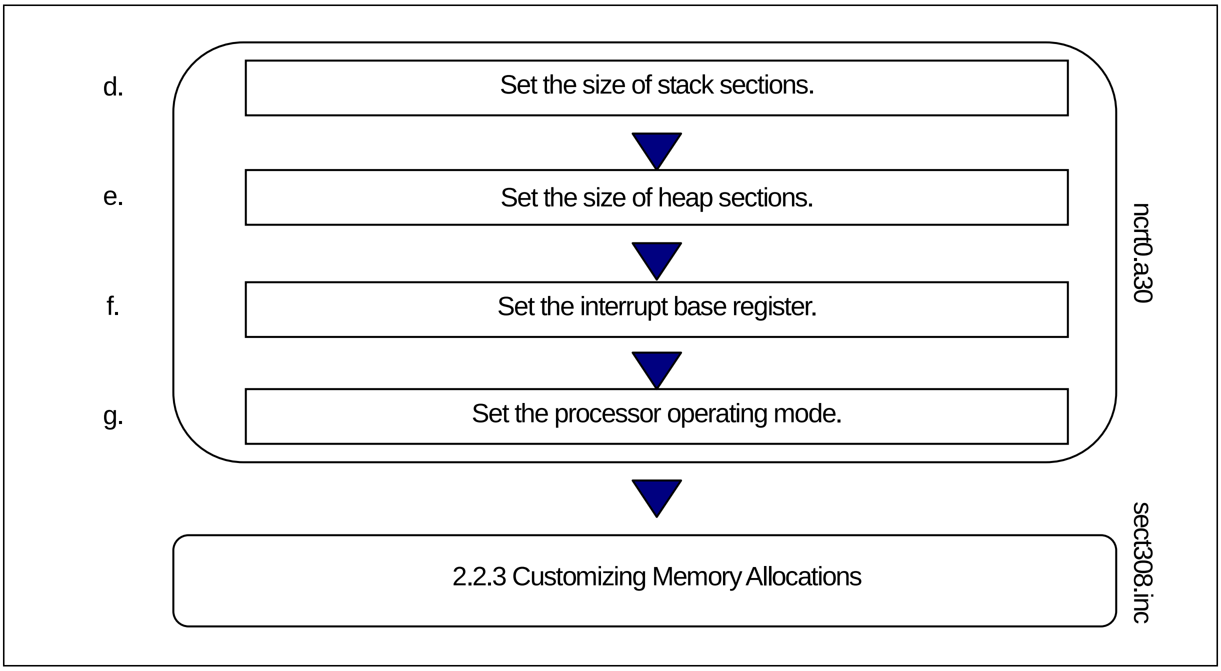


Figure 2.11 Example Sequence for Modifying Startup Programs

c. Examples of startup modifications that require caution

(1) Settings When Not Using Standard I/O Functions

The init function[3] initializes the M16C/80 Series I/O. It is called before main in ncrt0.a30.
Figure 2.12 shows the 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 2.12 Part of ncrt0.a30 Where init Function is Called

If you are using only sprintf and sscanf, the init function does not need to be called.

[3] The init function also initializes the microcomputer (hardware) for standard in-put/output functions. By default, the M16C/60 and the R8C Family is assumed to be the microcomputer that it initializes.
When using standard input/output functions, the init function, etc. may need to be modified depending on the system in which the microcomputer is to be used.

(2) Settings When Not Using Memory Management Functions

To use the memory management functions calloc and malloc, etc., not only is an area allocated in the heap section but the following settings are also made in ncrt0.a30.

(1) Initialization of external variable char *_mnext
Initializes the heap_top label, which is the starting address of the heap section.

(2) Initialization of external variable unsigned_msize
Initializes the "HEAPSIZE" expression, which sets at "2.2.2 e heap section size".

Figure 2.13 shows the initialization performed in ncrt0.a30.

```
;====================================================================
; heap area initialize
;---------------------------------------------------------------------
.if __HEAP__ != 0
        .glb        __mnext
        .glb        __msize
        mov.w       #(heap_top&0FFFFH), __mnext
        mov.w       #(heap_top>>16) __mnext+2
        mov.w       #(HEAPSIZE&0FFFFH), __msize
        mov.w       #(HEAPSIZE>>16), __msize+2
.endif
```

Figure 2.13 Initialization When Using Memory Management Functions (ncrt0.a30)

If your application program does not use memory-management functions, set the __HEAPSIZE__ macro within nc_define.inc to 0.This saves the ROM size by stopping unwanted library items from being linked.

(3) Notes on Writing Initialization Programs

Note the following when writing your own initialization programs to be added to the startup program.

(1) If your initialization program changes the U, or B flags, return these flags to the original state where you exit the initialization program. Do not change the contents of the SB register.

(2) If your initialization program calls a subroutine written in C, note the following two points:
- Call the C subroutine only after clearing them, B and D flags.
- Call the C subroutine only after setting the U flag.

d. Setting the Stack Section Size

A stack section has the domain used for user stacks, and the domain used for interruption stacks. Since stack is surely used, please surely secure a domain. stack size should set up the greatest size to be used.[4]
Stack size is calculated to use the stack size calculation utility Call Walker.

[4] The stack is used within the startup program as well. Although the initial values are reloaded before calling the main() function, consideration is required if the stack size used by the main() function, etc. is insufficient.

e. Heap Section Size

Set the heap to the maximum amount of memory allocated using the memory management functions calloc and malloc in the program. Set the heap to 0 if you do not use these memory management functions. Make sure that the heap section does not exceed the physical RAM area.

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

Figure 2.14 Example of Setting Heap Section Size (nc_define.inc)

f. Setting the interrupt vector table

Set the top address of the interrupt vector table to the part of Figure 2.15 in nc_defineinc. The INTB Register is initialized by the top address of the interrupt vector table.

```
__VECTOR_ADR__      .equ      0ffd00H             ; INTERRUPT VECTOR ADDRESS definition
__SPECIAL_PRG__     .equ 0f8000H                  ; Special page program address
```

Figure 2.15 Example of Setting Top Address of Interrupt Vector Table (nc_deinfe.inc)

The sample startup program has had values set for the tables listed below.

0FFD00H - 0FFDFFH:	Interrupt vector table
0FFE00H - 0FFFFFH:	Special page vector table and fixed vector table

Normally, these set values do not need to be modified.

g. Setting the Processor Mode Register

Set the processor operating mode to match the target system at address 04H (Processor mode register) in the part of ncrt0.a30 shown in Figure 2.16.

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

Figure 2.16 Example Setting of Processor Mode Register (ncrt0.a30)

See the User's Manual of microcomputer you are using for details of the Processor Mode Register.

2.2.3 Customizing for NC30 Memory Mapping

a. Structure of Sections

In the case of a native environment compiler, the executable files generated by the compiler are mapped to memory by the operating system, such as UNIX. However, with crossenvironment compilers such as this compiler, the user must determine the memory mapping.

With this compiler, storage class variables, variables with initial values, variables without initial values, character string data, interrupt processing programs, and interrupt vector address tables, etc., are mapped to Micoro Processor series memory as independent sections according to their function.

The names of sections consist of a base name and attribute as shown below:

Section Base Name	_	Attribute

Figure 2.17 Section Names

Table 2.11 shows Section Base Name and Table 2.12 shows Attributes.

Table 2.11 Section Base Names

Section base name	Content
data	Stores data with initial values
bss	Stores data without initial values
rom	Stores character strings, and data specified in #pragma ROM or with the const modifier

Table 2.12 Section Naming Rules

<table>
<tr><th>Attribute</th><th colspan="2">Meaning</th><th>Target section base name</th></tr>
<tr><td>I</td><td colspan="2">Section containing initial values of data</td><td>data</td></tr>
<tr><td rowspan="3">N/F/S</td><td>N</td><td>near attribute[5]</td><td rowspan="2">data, bss, rom</td></tr>
<tr><td>F</td><td>far attribute</td></tr>
<tr><td>S</td><td>SBDATA attribute</td><td>data, bss</td></tr>
<tr><td rowspan="2">E/O</td><td>E</td><td>Even data size</td><td rowspan="2">data, bss, rom</td></tr>
<tr><td>O</td><td>Odd data size</td></tr>
</table>

Table 2.13 shows the contents of sections other than those based on the naming rules described above.

Table 2.13 Section Names

Section name	Contents
fvector	This section stores the contents of the Micro Processor's fixed vector.
heap	This memory area is dynamically allocated during program execution by memory management functions (e.g., malloc). This section can be allocated at any desired location of the Micro Processor RAM area.
program	Stores programs
program_S	Stores programs for which #pragma SPECIAL has been specified.
stack	This area is used as a stack. Allocate this area at addresses between 0400H to 0FFFFH.
switch_table	The section to which the branch table for switch statements is allocated. This section is generated only with the "-fSOS" option.
vector	This section stores the contents of the Micro Processor's interrupt vector table. The interrupt vector table can be allocated at any desired location of the Micro Processor's entire memory space by intb register relative addressing. For more information, refer to the Micro Processor User's Manual.

These sections are mapped to memory according to the settings in the startup program include file sect30.inc. You can modify the include file to change the mapping.
Figure 2.18 shows the how the sections are mapped according to the sample startup program's include file sect30.inc.

[5] near and far are NC30 modifiers, used to clarify the addressing mode.
near........ accessible from 000000H to 00FFFFH
far.......... accessible from 000000H to 0FFFFFH

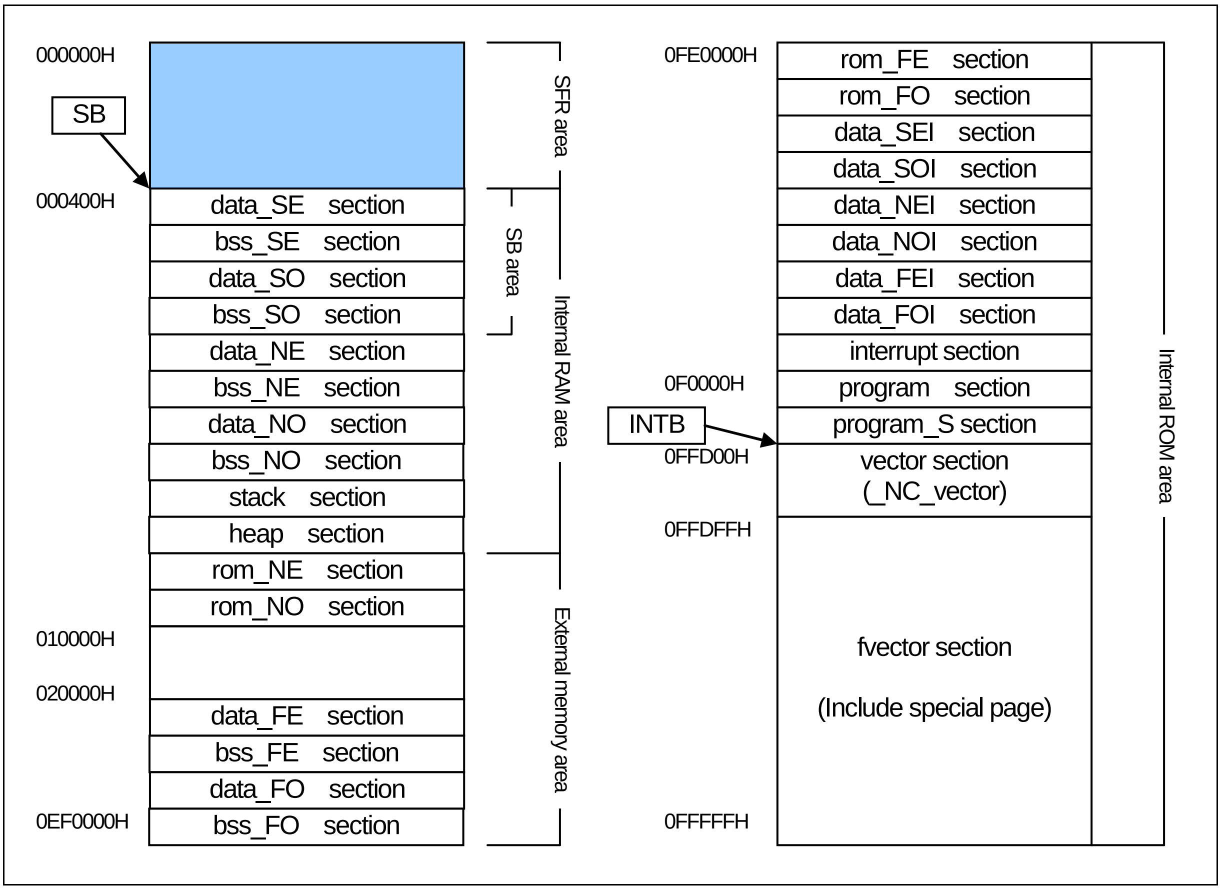


Figure 2.18 Example Section Mapping

b. Outline of memory mapping setup file

(1) About sect30.inc

This program is included from ncrt0.a30. It performs the following process mainly:

- Maps each section (in sequence)
- Sets the starting addresses of the sections
- Defines the size of the stack and heap sections
- Sets the interrupt vector table
- Sets the fixed vector table

c. Modifying the sect30.inc

Figure 2.19 summarizes the steps required to modify the startup programs to match the target system.

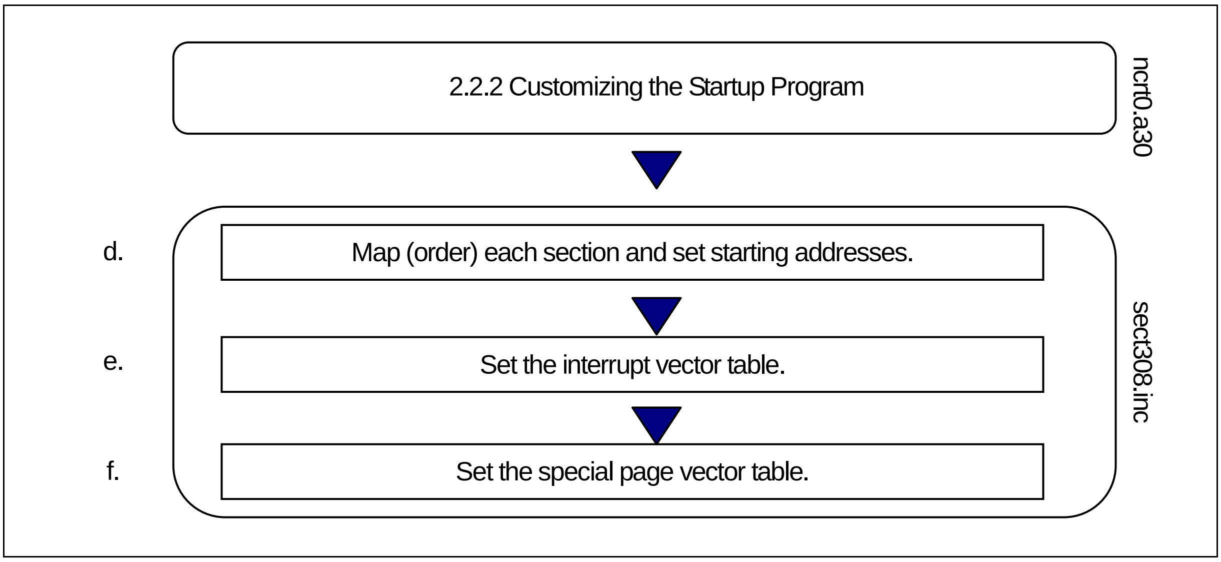


Figure 2.19 Example Sequence for Modifying Startup Programs

d. Mapping and Order Sections and Specifying Starting Address

Map and order the sections to memory and specify their starting addresses (mapping programs and data to ROM and RAM) in the sect30.inc include file of the startup program.

The sections are mapped to memory in the order they are defined in sect30.inc. Use the assembler pseudo instruction .ORG to specify their starting addresses.
Figure 2.20 is an example of these settings.

```
.section    program
.org                  0F0000H          ← Specifies the starting address of the program section
```

Figure 2.20 Example Setting of Section Starting Address

If no starting address is specified for a section, that section is mapped immediately after the previously defined section.

(1) Rules for Mapping Sections to Memory

Because of the effect on the memory attributes (RAM and ROM) of Micro Processor memory, some sections can only be mapped to specific areas. Apply the following rules when mapping sections to memory.

(1) Sections mapped to RAM

- stack section
- heap 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

Note also that some sections can only be mapped to specific memory areas in the Micro Processor memory space.

(1) Sections mapped only to 0H - 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 mapped only to 0F0000H - 0FFFFFH

- program_S section

(3) Sections mapped to any area.

- program section
- 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

If any of the following data sections have a size of 0, they need not be defined.

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

(2) Example Section Mapping in Single-Chip Mode

Figure 2.21 to Figure 2.24 are examples of the sect30.inc include file which is used for mapping sections to memory in single-chip mode.

```
;---------------------------------------------------------------
;
;               Arrangement of section
;
;---------------------------------------------------------------
; Near RAM data area
;---------------------------------------------------------------
; SBDATA area
                .section    data_SE,DATA
                .org        400H
data_SE_top:

                .section    bss_SE,DATA,ALIGN
bss_SE_top:

                .section    data_SO,DATA
data_SO_top:

                .section    bss_SO,DATA
bss_SO_top:

; near RAM area
                .section    data_NE,DATA,ALIGN
data_NE_top:

                .section    bss_NE,DATA,ALIGN
bss_NE_top:

                .section    data_NO,DATA
data_NO_top:

                .section    bss_NO,DATA
bss_NO_top:

;---------------------------------------------------------------
; Stack area
;---------------------------------------------------------------
                .section    istack,DATA,ALIGN
                .blkb                   __ISTACKSIZE__
istack_top:

.if __STACKSIZE__ != 0
                .section    stack,DATA,ALIGN
                .blkb                   __STACKSIZE__
stack_top:
.endif:
```

Figure 2.21 Listing of sect30.inc in Single-Chip Mode (1)

```
;---------------------------------------------------------------
;                heap section
;---------------------------------------------------------------
.if __HEAP__ != 0
                .section      heap,DATA
heap_top:
                .blkb         HEAPSIZE
.endif

;---------------------------------------------------------------
; Near ROM data area
;---------------------------------------------------------------
.if __NEAR_ROM_FLG__ != 0
                .section      rom_NE,ROMDATA,ALIGN
rom_NE_top:

                .section      rom_NO,ROMDATA
rom_NO_top:
.endif
;---------------------------------------------------------------
; Far RAM data area
;---------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                .section      data_FE,DATA
                .org                        10000H
data_FE_top:

                .section      bss_FE,DATA,ALIGN
bss_FE_top:

                .section      data_FO,DATA
data_FO_top:
                .section      bss_FO,DATA
bss_FO_top:
.endif
```

You can remove this part, because it is unnecessary.

In this case,you need to remove the initialize program in the far area of ncrt0.a30.

Figure 2.22 Listing of sect30.inc in Single-Chip Mode (2)

```
;---------------------------------------------------------------
; Far ROM data area
;---------------------------------------------------------------
                .section    rom_FE,ROMDATA
                .org                    __ROM_TOPADR__
rom_FE_top:

                .section    rom_FO,ROMDATA
rom_FO_top:

;---------------------------------------------------------------
; Initial data of 'data' section
;---------------------------------------------------------------
                .section    data_SEI,ROMDATA,ALIGN
data_SEI_top:

                .section    data_SOI,ROMDATA
data_SOI_top:

                .section    data_NEI,ROMDATA,ALIGN
data_NEI_top:

                .section    data_NOI,ROMDATA
data_NOI_top:

.if __FAR_RAM_FLG__ != 0
                .section    data_FEI,ROMDATA,ALIGN
data_FEI_top:

                .section    data_FOI,ROMDATA
data_FOI_top:
.endif
;---------------------------------------------------------------------
; Switch Table Section
;---------------------------------------------------------------------
;                           .section    switch_table,ROMDATA
;switch_table_top:
;---------------------------------------------------------------
; code area
;---------------------------------------------------------------
                            .section    program,CODE,ALIGN
                            .section    interrupt,CODE,ALIGN
                            .section    program_S,CODE,ALIGN
;---------------------------------------------------------------
; variable vector section
;---------------------------------------------------------------
                            .section    vector,ROMDATA
                            .org                    __VECTOR_ADR__
.if             0
                    :
                (omitted)
                    :
                .lword      dummy_int                       ; software int 63
.endif
```

Figure 2.23 Listing of sect30.inc in Single-Chip Mode (3)

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

; ROM code protect control address
                    ; .protect 00H
```

Figure 2.24 Listing of sect30.inc in Single-Chip Mode (4)

e. Setting Interrupt Vector Table

For programs that use interrupt processing, set up the interrupt vector table by one of the following two methods:

(1) Set up the interrupt vector table for the vector section in sect30.inc.

The content of the interrupt vector varies with each type of microcomputer, and must therefore be set up to suit the type of microcomputer used.

For details, refer to the user's manual included with your microcomputer.

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

For programs that use interrupt processing, change the interrupt vector table for the vector section in sect30.inc.

Figure 2.25 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 trance (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 trance/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 2.25 Interrupt Vector Address Table

The contents of the interrupt vectors varies according to the machine in the M16C/60 series and R8C Family. See the User Manual for your machine for details.

Change the interrupt vector address table as follows:

(1) Externally declare the interrupt processing function in the .GLB as30 pseudo instruction.

(2) The labels of functions created by NC30 are preceded by the underscore (_). Therefore, the names of interrupt processing functions declared here should also be preceded by the underscore.

(3) Replace the names of the interrupt processing functions with the names of interrupt processing functions that use the dummy interrupt function name dummy_int corresponding to the appropriate interrupt table in the vector address table.

Figure 2.26 is an example of registering the UART1 send interrupt processing function uarttrn.

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

(omitted)
```

Figure 2.26 Example Setting of Interrupt Vector Addresses

Chapter 3 Programming Technique

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

3.1 Notes

Renesas Electronics Corp. 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 Corp., 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 NC30 vary in contents depending on the startup options specified when compiling, contents of version changes, etc. Therefore, when you have changed the startup options or upgraded the compiler version, be sure to reevaluate the operation of your application program.
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.
In this compiler, the instructions which cannot be used may be generated for writing and read-out to the register of SFR area. When you describe like the following examples as C language description to a SFR area, in this compiler may generate the assembler code which carries out operation which is not assumed since the interrupt request bit is not normal.
When accessing registers in the SFR area in 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 */
}
```

Figure 3.1 C language description to 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;
}
```

Figure 3.2 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 Figure 3.3

```
extern int volatile     port;

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

Figure 3.3 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;
}
```

Figure 3.4 meaningless Comparison

In the case of this example, because the variable c is written as char, the compiler treats it as the unsigned char type. Since the range of values re-presentable by the unsigned char type is 0 to 255, the variable c will never take on the value -1.
Accordingly, if there is any statement which logically has no effect like this example, the compiler does not generate assembler code.

(3) Programs not executed

No assembler codes are generated for programs which logically are not executed.

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

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

Figure 3.5 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;
}
```

Figure 3.6 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? */
a = ++b;            /* whether a = b or a = (b + 1)? */
```

Figure 3.7 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.
Because improper use of this facility produces negative effects, always be sure to examine generated codes before deciding to use it.

b. About register qualification and optimization options

When optimization options are specified, variables are assigned to registers as one optimization feature. This assignment feature is not affected 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, when it is known that the array size does not exceed 64K bytes, explicitly state the size in extern declaration of far type array as shown in Figure 3.8 or add the compile option "-fsmall_array(-fSA)"[2] before compiling. This helps to increase the code efficiency of the program.

```
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.
```

Figure 3.8 Example extern-Declaration of far Array

[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.

c. Using Prototype declaration Efficiently

NC30 allows you to accomplish an efficient function call by declaring the prototype of a function.
This means that unless a function is declared of its prototype in NC30, arguments of that function are saved on the stack following the rules listed in Table 3.1 when calling the function.

Table 3.1 Rules for Using Stack for Parameters

Data type(s)	Rules for saving on stack
char signed char	Expanded into the int type when stacked.
float	Expanded into the double type when stacked.
otherwise	Not expanded when stacked.

For this reason, NC30 may require redundant type expansion unless you declare the prototype of a function.

Prototype declaration of functions helps to suppress such redundant type expansion and also makes it possible to assign arguments to registers. All this allows you to accomplish an efficient function call.

d. Using SB Register Efficiently

Using the SB register-based addressing mode, you can reduce the size of your application program (ROM size). NC30 allows you to declare variables that use the SB register-based addressing mode by declaring #pragma SBDATA as shown in Figure 3.9.

```
#pragma SBDATA   val

int         val;
```

Figure 3.9 Example of variable declaration using SB-based addressing mode

e. Compressing ROM Size Using Compile Option -fJSRW

When calling a function defined outside the file in NC30, the function is called with the JSR.A instruction. However, if the program is not too large, most functions can be called with the "JSR.W" instruction.
In this case, ROM size will be reduced by doing as follows :
First, Compile with the -fJSRW option and check functions which are indicated as errors at link-time. Then change declarations for the error functions only into declarations using "#pragma JSRA function-name".
When you use the OGJ option, the JMP instruction at the time of a link is chosen.

f. 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.

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 Figure 3.10 does not initialize the variable, which must therefore be initialized to 0 (by clearing the bss[3] area) during the startup routine.

```
static int     i;
```

Figure 3.10 Example Declaration of Variable Without Initial Value

In some instances, it is not necessary for a variable with no initial value to be cleared to 0. In such cases, you can comment out the routine for clearing the bss area in the startup program to increase the speed of startup processing.

```
;====================================================================
; NEAR area initialize.
;--------------------------------------------------------------------
; bss zero clear
;--------------------------------------------------------------------
;               BZERO     bss_SE_top,bss_SE
;               BZERO     bss_SO_top,bss_SO
;               BZERO     bss_NE_top,bss_NE
;               BZERO     bss_NO_top,bss_NO
                    :
                (omitted)
                    :
;====================================================================
; FAR area initialize.
;--------------------------------------------------------------------
; bss zero clear
;--------------------------------------------------------------------
;               BZERO     bss_SE_top,bss_SE
;               BZERO     bss_SO_top,bss_SO
```

Figure 3.11 Commenting Out Routine to Clear bss Area

[3] The external variables in RAM which do not have initial values are referred to as "bss".

3.3 Linking Assembly Language Programs with C Programs

3.3.1 Calling Assembler Functions from C Programs

a. Calling Assembler Functions

Assembler functions are called from C programs using the name of the assembler function in the same way that functions written in C would be.
The first label in an assembler function must be preceded by an underscore (_). However, when calling the assembly function from the C program, the underscore is omitted. The calling C program must include a prototype declaration for the assembler function.
Figure 3.12 is an example of calling assembler function asm_func.

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

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

Figure 3.12 Example of Calling Assembler Function Without Parameters(sample.c)

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

Figure 3.13 Compiled result of sample.c(sample.a30)

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) Write a prototype declaration for the assembler function before the #pragma PARAMETER declaration. You must also declare the parameter type(s).

(2) Declare the name of the register used by #pragma PARAMETER in the assembler function's parameter list.

Figure 3.14 is an example of using #pragma PARAMETER when calling the assembler function asm_func.

```
extern unsigned int     asm_func(unsigned int, unsigned int);
#pragma PARAMETER    asm_func(R0, R1)             ← 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);
}
```

Figure 3.14 Example of Calling Assembler Function With Parameters (sample2.c)

```
                .SECTION  program,CODE,ALIGN
                ._file    'sample2.c'
                .align
                ._line    5
;## # C_SRC :             {
                .glb      _main
_main:
                enter     #04H
                ._line    6
;## # C_SRC :                       int         i = 0x02;
                mov.w     #0002H,-4[FB]         ;   i
                ._line    7
;## # C_SRC :                       int         j = 0x05;
                mov.w     #0005H,-2[FB]         ;   j
                ._line    9
;## # C_SRC :                       asm_func(i, j);
                mov.w     -2[FB],R1   ;   j              ← Parameters are passed via the R0 and R1
                mov.w     -4[FB],R0   ;   i                registers to the assembler function.
                jsr       _asm_func
                ._line    10
;## # C_SRC :             }
                exitd
E1:
                .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.
```

Figure 3.15 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 (flong longparameters
- Floating point type (double) parameters and long double 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 following shows a procedure for writing the entry processing of assembler functions.

(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) When modifying the B and U flags within the function, save the flag register to the stack beforehand.[4]

The following shows a procedure for writing the exit processing of assembler functions.

(5) If you modified the B and U flags within the function, restore the flag register from the stack.
(6) Write the RTS instruction.

Do not change the contents of the SB and FB registers in the assembler function. If the contents of the SB and FB registers are changed, save them to the stack at the entry to the function, then restore their values from the stack at the exit of the function.

Figure 3.16 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          ← (1)
          .glb        _asm_func        ← (2)
_asm_func:                             ← (3)
          pushc       FLG              ← (4)
          mov.w       SYM1, R1
          mov.w       SYM1+2, R3

          popc        FLG              ← (5)
          rts                          ← (6)
          .END
```

Figure 3.16 Example Coding of Assembler Function

[4] Do not change the contents of B and U flags in the 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. Figure 3.17 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	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       #0A000H, R0
            mov.w       #0001H, R2
            rts
            .END
```

Figure 3.17 Example of Coding Assembler Function to Return long-type Return Value

c. Referencing C Variables

Because assembler functions are written in different files from the C program, only the C global variables can be referenced.

When including the names of 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.
Figure 3.18 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
```

Figure 3.18 Referencing a C Global Variable

d. Notes on Coding Interrupt Handing in Assembler Function

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.

Figure 3.19 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
```

Figure 3.19 Example Coding of Interrupt Processing Assembler Function

e. Notes on Calling C Functions from Assembler Functions

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

(1) Call the C function using a label preceded by the underscore (_) or the dollar ($).
(2) The contents of the registers in use are not saved at the entry point of C-language functions. For this reason, you need to save these contents before calling a C-language function from an assembly-language program.

3.3.3 Notes on Coding Assembler Functions

Note the following when writing assembly language functions (subroutines) that are called from a C program.

a. Notes on Handling B and U flags

When returning from an assembler function to a 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 you modified the FB (frame base) register in an assembler function, you may not be able to return normally to the C language program from which the function was called.

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.
Figure 3.20 shows the format (asm_func in the figure is the name of an assembler function).

```
unsigned int    asm_func(unsigned int, unsigned int);          ← Prototype declaration of assembler function

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

Figure 3.20 Prototype declaration of assembler function

#pragma PARAMETER passes arguments to assembler functions via 16-bit generalpurpose registers (R0, R1, R2, R3), 8-bit general-purpose registers (R0L, R0H, R1L, R1H), and address registers (A0, A1). In addition, the 16-bit general-purpose registers are combined to form 32-bit registers (R3R1 and R2R0) for the parameters to be passed to the Note that an assembler function's prototype must always be declared before the #pragma PARAMETER declaration.
However, you cannot declare the following parameter types in a #pragma PARAMETER declaration:

- struct or union types
- 64bit integer type (flong longparameters
- floating point type(double) argument and long double argument

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 nc30 and the command line options. The description of the command line options includes those for the as30 assembler and ln30 linkage editor, which can be started from nc30.

A.1 nc30 Command Format

```
% nc30∆[command-line-option]∆[assembly-language-source-file-name]∆
                                      [relocatable-module-file-name]∆<C-source-file-name>

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

Figure A.1 nc30 Command Line Format

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

<RET> : Return key
* Always specify the startup program first when linking.
```

Figure A.2 Example nc30 Command Line

A.2 nc30 Command Line Options

A.2.1 Options for Controlling Compile Driver

Table A.1 shows the command line options for controlling the compile driver.

Table A.1 Options for Controlling Compile Driver

Option	Function
-c	Creates a relocatable module file (extension .r30) and ends processing [1]
-D*dentifier*	Defines an identifier. Same function as #define.
-dsource (Short form -dS)	Generates an assembly language source file (extension ".a30") with a C language source list output as a comment. (Not deleted even after assembling.)
-dsource_in_list (Short form -dSL)	In addition to the "-dsource(-dS)" function, generates an assembly language list file (.lst).
-E	Invokes only preprocess commands and outputs result to standard output.
-I*directory*	Specifies the directory containing the file(s) specified in #include. You can specify up to 256 directories.
-P	Invokes only preprocess commands and creates a file (extension .i).
-S	Creates an assembly language source file (extension .a30) and ends processing.
-silent	Suppresses the copyright message display at startup.
-U*predefined macro*	Undefines the specified predefined macro.

-c

Compile driver control

Function: Creates a relocatable module file (extension .r30) and finishes processing.

Notes: If this option is specified, no absolute module file (extension .x30) or other file output by ln30 is created.

-D*dentifier*

Compile driver control

Function: The function is the same as the preprocess command #define.
Delimit multiple identifiers with spaces.

Syntax: nc30∆-D*identifier*[=*constant*]∆<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.

[1] If you do not specify command line options -c, -E, -P, or -S, nc30 finishes at and output files up to the absolute load module file (extension .x30) are created.

-dsource -dS

Comment option

Function: Generates an assembly language source file (extension ".a30") with a C language source list output as a comment (Not deleted even after assembling).

Supplement:
(1) -When the -S option is used, the option "-dsouce(-dS)" is automatically enabled.
(2) The generated files ".a30" and ".r30" are not deleted. Use this option when you want to output C-language source lists to the assembly list file.

-dsource_in_list -dSL

List File option

Function: In addition to the "-dsource(-dS)" function, generates an assembly language list file (filename extension ".lst").

-E

Compile driver control

Function: Invokes only preprocess commands and outputs results to standard output.

Notes: When this option is specified, no assembly source file (extensions .a30), relocatable module files (extension .r30), absolute module files (extension .x30), or other files output by ccom30, as30, or ln30 are generated.

-I*directory*

Compile driver control

Function: Specifies the directory name in which to search for files to be referenced by the preprocess command #include.
Max specified 256 directory.

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: nc30Δ-I*directory*Δ<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 specified, no assembly source file (extensions .a30), relocatable module files (extension .r30), absolute module files (extension .x30) or other files output by ccom30, as30, or ln30 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, try again with the -E option.

-S

Compile driver control

Function: Creates assembly language source files (extension .a30 and .ext) and stops processing.

Notes: When this option is specified, no relocatable module files (extension.r30), absolute module files (extension .x30) or other files output by as30 or ln30 are generated.

-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: nc30Δ-U*predefined macro*Δ<C source file>

Notes: The maximum number of macros that can be undefined 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.
STDC, _LINE_, _FILE_, _DATE_, and _TIME_ cannot be undefined.

A.2.2 Options Specifying Output Files

Table A.2 shows the command line option that specifies the name of the output machine language data file.

Table A.2 Options for Specifying Output Files

Option	Function
-dir*directory-name*	Specifies the destination directory of the file(s) (absolute module file, map file, etc.) generated by ln30.
-o*file-name*	Specifies the name(s) of the file(s) (absolute module file, map file, etc.) generated by ln30. This option can also be used to specify the destination directory. This option can also be used to specify the file name includes the path. Do not specify the filename extension.

-dir *directory-name*

Output file specification

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

Syntax: nc30Δ-dir*directory-name*

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 name(s) of the file(s) (absolute module file, map file, etc.) generated by ln30.
This option can also be used to specify the file name includes the path.
You must not specify the filename extension.

Syntax: nc30Δ-o*file-name*Δ<C source file>

A.2.3 Version Information Display Option

Table A.3 shows the command line options that display the cross-tool version data.

Table A.3 Options for Displaying Version Data

Option	Function
-v	Displays the name of the command program and the command line during execution.
-V	Displays the startup messages of the compiler programs, then finishes processing (without compiling).

-v

Display command program name

Function: Compiles the files while displaying the name of the command program that is being executed.

Notes: Use lowercase v for this option.

-V

Display version data

Function: Displays version data for the command programs executed by the compiler, then finishes processing.

Supplement: Use this option to check that the compiler has been installed correctly. The "M16C Family C Compiler package Release Notes" list the correct version numbers of the commands executed internally by the compiler.

If the version numbers in the Release Notes do not match those displayed using this option, the package may not have been installed correctly. See the "M16C Family C Compiler package Release Notes" for details of how to install the NC30 package.

Notes:
(1) Use uppercase V for this option.
(2) If you specify this option, all other options are ignored.

A.2.4 Options for Debugging

Table A.4 shows the command line options for outputting the symbol file for the C source file.

Table A.4 Options for Debugging

Option	Function
-g	Outputs debugging information to an assembler source file (extension.a30).Therefore you can perform C language-level debugging.
-genter	Always outputs an enter instruction when calling a function. Be sure to specify this option when using the debugger's stack trace function.
-gno_reg	Suppresses the output of debugging information for register variables.
-gbool_to_char	This option outputs bool-type debugging information as the char type.
-gold	This option outputs debugging information in Rev. E format. When this option specifies, the "-gno_reg" option and the "-fauto_128" option are automatically specified.

-g

Outputting debugging information

Function: Outputs debugging information to an assembler source file (extension .a30).

Notes: When debugging your program at the C language level, always specify this option. Specification of this option does not affect the code generated by the compiler.
When "-finfo" option is specified, this option becomes effective.

-genter

Outputting enter instruction

Function: Always output an enter instruction when calling a function.

Notes:
(1) When using the debugger's stack trace function, always specify this option. Without this option, you cannot obtain the correct result.
(2) When this option is specified, the compiler generates code to reconstruct the stack frame using the enter command at entry of the function regardless of whether or not it is necessary. Consequently, the ROM size and the amount of stack used may increase.

-gno_reg

Suppresses debugging information about register variables

Function: Suppresses the output of debugging information for register variables.

Supplement: Use this option to suppress the output of debugging information about register variables when you do not require that information. Suppressing the output of debugging information about the register variables will speed up downloading to the debugger.

-gbool_to_char -gBTC

Outputting debugging information

Function: This option outputs bool-type debugging information as the char type.

Supplement: This option is necessary if you are using an old PDB30 that does not support the bool type.

-gold

Outputs debugging information in previous format

Function: This option outputs debugging information in Rev.E format.
When this option specifies, the "-gno_reg" option and the "-fauto_128" option are automatically specified.

Supplement: With the increase in the maximum number of auto variables, starting with NC30 V.2.00, the format of debugging information has changed(from xxx.r30 and xxx.x30 format). The new format is known as the Rev. F format. the executable objects in the new format(xxx.x30) are compatible with the following debuggers:
(1) PDB30 V.2.00 and later
(2) PDB30SIM V.2.00 and later
(3) High-performance Embedded Workshop V.4.00 and later
Use the -gold option when compiling if you are using a debugger that cannot load executable objects in the new format (xxx.x30).

A.2.5 Optimization Options

Table A.5 shows the command line options for optimizing program execution speed and ROM capacity.

Table A.5 Optimization Options

Option	Short form	Function
-O[1-5]	None	Optimization of speed and ROM size.
-OR	None	Optimization of ROM size.
-OS	None	Optimization of speed.
-OR_MAX	-ORM	Places priority on ROM size for the optimization performed.
-OS_MAX	-OSM	Places priority on for the optimization performed.
-Ocompare_byte_to_word	-OCBTW	Compares consecutive bytes of data at contiguous addresses in words.
-Oconst	-OC	Performs optimization by replacing references to the const-qualified external variables with constants.
-Oforward_function_to_inline	-OFFTI	Expands all inline functions in-line.
-Oglb_jmp	-OGJ	Global jump is optimized.
-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	Inhibits starting 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 data.
-Ono_float_const_fold	-ONFCF	Suppresses the constant folding processing of floating point numbers.
-Ono_logical_or_combine	-ONLOC	Suppresses the optimization that puts consecutive OR 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 allows the necessary ROM capacity to be reduced. However, this may result in an increased amount of stack being used. Please specify this option with –O[1-5].
-Ostack_frame_align	-OSFA	Aligns the stack frame on an even boundary.
-Ostatic_to_inline	-OSTI	A static function is treated as an inline function.
-O5OA	None	Inhibits code generation based on bit-manipulating instructions when the optimization option "-O5" is selected.

The effects of main optimization options are shown in Table A.6.

Table A.6 Effect of each Optimization Options

Option	-O	-OR	-OS	-OSA	-OSFA
SPEED	faster	lower	faster	faster	faster
ROM size	decrease	decrease	increase	decrease	Same[2]
usage of stack	decrease	increase	same	increase	increase

[2] -OSFA adjust address of stacks of each function entry to an even-numbered address. Therefore, if a function has no auto variable declaration, because enter #00H is always added, the processing

-O[1-5]

Optimization

Function:

Optimizes speed and ROM size. This option can be specified with -g options.-O3 is assumed if you specify no numeric (no level).

-O1: Some representative optimization items executed by this option are the following.
- Allocate the register the variable.
- Delete a meaningless conditional expression.
- Deletion of statement not logically executed.

-O2: Makes no difference with "-O1".

-O3: Execute some optimization items addition to "-O1".
Some representative optimization items executed by this option are the following.
- Grouping of bit manipulations.
- Constant folding processing of floating point numbers.
- Inline padding of standard library functions.

-O4: Execute some optimization items addition to "-O3".
Some representative optimization items executed by this option are the following.
- Replace the reference to the variable declared in the const-qualifier with constants.

-O5: Execute some optimization items addition to "-O4".
Some representative optimization items executed by this option are the following.
- Optimization of address computations such as pointers and structures(if the option "-OR" is concurrently specified).
- Strengthen the optimization of the pointer(if the option "-OS" is concurrently specified).

However, a normal code may be unable to be outputted when fulfilling the following conditions.

- With a different variable points out the same memory position simultaneously within a single function and they point to an-identical address.

```
Example:

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

void        test1(void)
{
            int         b;
            *p = 9;
            a = 10;
            b = *p;                 /* By applying optimization, "p" will be transposed to "9". */
            printf( "b = %d (expect b = 10)¥n",b );
}

result:

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

-O[1-5]

Optimization

Notes:

When the "-O5" optimizing options is used, the compiler generates in some cases "BTSTC" or "BTSTS" bit manipulation instructions. In M16C, the "BTSTC" and "BTSTS" bit manipulation instructions are prohibited from rewriting the contents of the interrupt control registers.

However, the compiler does not recognize the type of any register, so, should "BTSTC" or "BTSTS" instructions be generated for interrupt control registers, the assembled program will be different from the one you intend to develop.

When the "-O5" optimizing options is used in the program shown below, a "BTSTC" instruction is generated at compilation, which prevents an interrupt request bit from being processed correctly, resulting in the assembled program performing improper operations.

C sauce which must not use an optimization option at the time of compile:

```
#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 */
}
```

Please compile after taking the following measures, if the manipulation instructions is generated to bit operation of SFR area. Make sure that no "BTSTC" and "BTSTS" instructions are generated after these side-steppings.

- Optimization options other than "-O5" are used.
- An instruction is directly described in a program using an ASM function
- Add the "-O5OA" option.

-OR

Optimization

Function:

Optimizes ROM size in preference to speed. This option can be specified with "-g" and "-O" options.

Notes:

When this option is used, the source line information may partly be modified in the course of optimization. Therefore, if this options is specified, when your program is running on the debugger, your program is a possibility of different actions.
If you do not want the source line information to be modified, use the "-One_break_source_debug(-ONBSD)" option to suppress optimization.

-OS

Optimization

Function: Although the ROM size may somewhat increase, optimization is performed to obtain the fastest speed possible.
This option can be specified along with the "-g" option.

-OR_MAX -ORM

Optimization

Function: Places priority on ROM size for the optimization performed.

Explanation:

(1) The compile options listed below are enabled.

- -O5
- -OR
- -O5OA
- -Oglb_jmp (-OGJ)
- -fchar_enumerator (-fCE)
- -fdouble_32 (-fD32)
- -fno_align (-fNA)
- -fno_carry (-fNC)
- -fsmall_array (-fSA)
- -fuse_DIV (-fUD)

(2) If this option is used in the integrated development environment or High-performance Embeded Workshop, be sure to enable "Size or speed:" on the C tab of the Renesas M16C Standard Toolchain and then select ""ROM size to the minimum".

Notes:

(1) The source line information may partly be modified in the course of optimization. Therefore, if this options is specified, when your program is running on the debugger, your program is a possibility of different actions. If you do not want the source line information to be modified, use the compile option "-One_break_source_debug(-ONBSD)" to suppress optimization.

(2) Please make sure to specify link option "-JOPT".

(3) The enum type may not be referenced correctly in some debugger.

(4) A function prototype must always be expressly written. Without a prototype declaration, the compiler may not be able to generate the correct code.

(5) The debug information of the type double is processed as the type float. So, the data of the type double is displayed as the type float on C watch window and global window of Debug tool.

(6) When far-type pointers are used to indirectly access memory dynamically allocated using the malloc function, etc., or ROM data mapped to the far area, be sure that the data is not accessed spanning a 64K bytes boundary.

(7) The function of compiler option "- fno_carry(-fNC)" is invalid when it is used in conjunction with "- R8C" or "- R8CE".

(8) If the divide operation results in an overflow, the compiler may operate differently than stipulated in ANSI.

-OS_MAX -OSM

Optimization

Function: Places priority on speed for the optimization performed.

Explanation:

(1) The compile options listed below are enabled.

- -O4
- -OS
- -Oforward_function_to_inline(-OFFTI)
- -Oglb_jmp (-OGJ)
- -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) If this option is used in the integrated development environment or High-performance Embeded Workshop, be sure to enable "Size or speed:" on the C tab of the Renesas M16C Standard Toolchain and then select ""ROM size to the minimum".

Notes::

(1) Please make sure to specify link option "-JOPT".
(2) The ROM size increases for reasons that the for statement is revolved.
(3) The assembler code to description of substance of the static function which became inline function treatment is always generated.
(4) About a function, it is compulsorily. In treating as an inline function, it is in a function. Please make an inline declaration.
(5) The enum type may not be referenced correctly in some debugger.
(6) A function prototype must always be expressly written. Without a prototype declaration, the compiler may not be able to generate the correct code.
(7) The debug information of the type double is processed as the type float. So, the data of the type double is displayed as the type float on C watch window and global window of Debug tool.
(8) When far-type pointers are used to indirectly access memory dynamically allocated using the malloc function, etc., or ROM data mapped to the far area, be sure that the data is not accessed spanning a 64K bytes boundary.
(9) The function of compiler option "- fno_carry(-fNC)" is invalid when it is used in conjunction with "- R8C" or "- R8CE".
(10) If the divide operation results in an overflow, the compiler may operate differently than stipulated in ANSI.
(11) When specifying an inline storage class for a function, be sure that inline storage class and this body definition are written in the same file where the function is written.
(12) No structures or unions can be used as the parameter of an inline function. If this is attempted, a compilation error occurs.
(13) No indirect calls of an inline function can be made. If this is attempted, a compilation error occurs.
(14) No recursive calls of an inline function can be made. If this is attempted, a compilation error occurs.

-Ocompare_byte_to_word -OCBTW

Optimization

Function: Compares consecutive bytes of data at contiguous addresses in words.

Notes: This option is only valid if you specify option -O[1 to 5], -OR, -OR_MAX(-ORM), -OS or -OS_MAX(-OSM)).

-Oconst -OC

Optimization

Function: Optimizes code generation by replacing reference to variables to declared by the const-qualifier with constants.
This is effective even when other than the "-O4" option is specified.

Supplement: Optimization is performed when all of the following conditions are met:

(1) Variables not including bit-fields and unions.
(2) Variables for which the const-qualifier is specified but are not specified to be volatile.
(3) Variables that are subject to initialization in the same C language soirce file.
(4) Variablew that are initialized by constant or const-qualified variables.

-Oforward_function_to_inline -OFFTI

Optimization

Function: Expands all inline functions in-line.

Supplement: Although inline functions require that an inline function be called after its entity definition can be made, use of this option allows the entity definition of an inline function to be made after calling it.

Notes:

(1) When specifying inline storage class for a function, be sure that inline storage class and this body definition is written in the same file as the function is written.
(2) The parameter of an in line function cannot be used by “structure” and "union" .It becomes a compile error.
(3) The indirect call of an in line function cannot be carried out. It becomes a compile error when a indirect call is described.
(4) The recursive call of an in line function cannot be carried out. It becomes a compile error when a recursive call is described.

-Oglb_jmp -OGJ

Optimization

Function: Global jump is optimized.

Notes: When you use this function, please make sure to specify link option “JOPT”

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

Unrolls 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 count of up to 5.

Supplement: Unrolled code is output for only the "for" statements where the number of times they are executed is known. Specify the upper-limit count for which times for is revolved in the target for statement to be unrolled.
By default, this option is applied to the for statements where for is revolved up to five times.

Notes: The ROM size increases for reasons that the for statement is revolved.

-Ono_asmopt -ONA

Inhibits starting the assembler optimizer

Function: Inhibits starting the assembler optimizer "aopt30".

-Ono_bit -ONB

Suppression of optimization

Function: Suppresses optimization based on grouping of bit manipulations.

Supplement: When you specify -O (or -OR or -OS or -OR_MAX or -ORM or -OS_MAX or -OSM), optimization is based on grouping manipulations that assign constants to a bit field mapped to the same memory area into one routine.
Because it is not suitable to perform this operation when there is an order to the consecutive bit operations, as in I/O bit fields, use this option to suppress optimization.

Notes: This option is only valid if you specify option -O[3 to 5], -OR or -OS or -OR_MAX or -ORM or -OS_MAX or -OSM.

-Ono_break_source_debug -ONBSD

Suppression of optimization

Function: Suppresses optimization that affects source line data.

Supplement: Specifying the " -OR" or "-O" option performs the following optimization, which may affect source line data. This option ("-ONBSD") is used to suppress such optimization.

Notes: This option is only valid if you specify option -O[3 to 5], -OR or -OS or -OR_MAX or -ORM or -OS_MAX or -OSM.

-Ono_float_const_fold -ONFCF

Suppression of optimization

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

Supplement: By default, NC30 folds constants. Following 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 points, the results of calculation differ as the order of calculation is changed. This option suppresses the constant folding in floating point numbers so that the calculation sequence in the C source file is preserved.

-Ono_logical_or_combine -ONLOC

Suppression of optimization

Function: Suppresses the optinization that puts consective ORs together.

Supplement: If one of three options "-O3 or greater, -OR, or -OS" is specified when compiling as in the example shown below, the compiler optimizes code generation by combining logical ORs.

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

In this case, the variable a is referenced up to three times, but after optimization it is referenced only once.
However, if the variable a has any effect on I/O references, etc., the program may become unable to operate correctly due to optimization. In such a case, specify this option to suppress the optimization to combine logical ORs. Note, however, that if the variable is declared with volatile, logical ORs are not combined
for optimization.

-Ono_stdlib -ONS

Suppression of optimization

Function: Suppresses inline padding of standard library functions, modification of library functions, and similar other optimization processing.

Supplement: This option suppresses the following optimization.

- Optimization for replacing the standard library functions such as "strcpy()" and "memcpy()" with the SMOVF instructions, etc.
- Optimization for changing to the library functions that conform to the arguments near and far.

Notes: Specify this option, when make a function which name is same as standard library function.

-Osp_adjust -OSA

Removing stack correction code after calling a function

Function: Optimizes code generation by combining stack correction codes after function calls.

Supplement: Because the area for arguments to a function normally is deallocated for each function call made, processing is performed to correct the stack pointer.
If this option is specified, processing to correct the stack pointer is performed collectively, rather than for each function call made.

Example:

In the example shown below, the stack pointer is corrected each time func1() and then func2() is called, so that the stack pointer is corrected twice. If this option is specified, 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: Use of the option "-Osp_adjust" helps to reduce the ROM capacity and at the same time, to speed up the processing. However, the amount of stack used may increase.
Please specify this option with -O[1-5], -OR, or -OS.

-Ostack_frame_align -OSFA

Aligns stack frame

Function: Aligns the stack frame on an even boundary
In the entry version, this option cannot be specified.

Supplement: When even-sized auto variables are mapped to odd addresses, memory access requires one more cycle than when they are mapped to even addresses. This option maps even-sized auto variables to even addresses, thereby speeding up memory access.

Notes:

(1) The following functions specified in #pragma are not aligned.
- #pragma INTHANDLER
- #pragma HANDLER
- #pragma ALMHANDLER
- #pragma CYCHANDLER
- #pragma INTERRUPT[3]

(2) Be sure that the stack point is initialized to an even address in the startup program. Also, be sure to compile all programs using this option.

[3] In order that there may be no guarantee the number of whose values of the stack pointer in the timing which interruption generated is even, alignment is not performed to an interruption function. For this reason, processing speed may become slow when "-Ostack_frame_align" option is specified to the function called from an interruption function.

-Ostatic_to_inline -OSTI

A static function is treated as an inline function

Function: A static function is treated as an inline function and the assembling code which carried out inline deployment is generated.

Supplement: When the following conditions are fulfilled, a static function is treated as an inline function and the assembling code which carried out inline deployment is generated.

(1) Substance is described before the function call. It is aimed at a static function.
- A function call and the body of that function must be written in the same source file.
- When you specify "-Oforward_function_to_inline" or "-OFFTI" option, ignore this condition.

(2) When address acquisition is omitted in the program to the static function.

(3) When the recursive call of the static function has not been carried out.

(4) When construction of a frame (reservation of an auto variable etc.) is not performed in the assembling code output of a compiler.
- The situation of the existence of frame construction changes with combined use with the contents of description of the target function, and another optimization option.
- When you specify "-Oforward_function_to_inline" or "-OFFTI" option, ignore this condition.

Below, inline deployment is carried out. The example of description of a static function is shown.

```
extern int      i;

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

void    main(void)
{
        int     s;

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

Function func() is a function.
inline deployment is carried out in each place currently called within main().

Notes:

(1) The assembler code to description of substance of the static function which became inline function treatment is always generated.

(2) About a function, it is compulsorily. In treating as an inline function, it is in a function. Please make an inline declaration.

-O5OA

Inhibit code generation

Function: Inhibits code generation based on 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 read or write to the registers in the SFR area. Select this option if when the optimization option "-O5" is selected codes are generated using bit-manipulating instructions for read or write to the registers in the SFR area.

A.2.6 Generated Code Modification Options

Table A.7 to Table A.8 shows the command line options for controlling nc30-generated assembly code.

Table A.7 Generated Code Modification Options (1)

Option	Short form	Function
-fansi	None	Makes "-fnot_reserve_far_and_near", "-fnot_reserve_asm", and "-fextend_to_int" valid.
-fchar_enumerator	-fCE	Handles the enumerator type as an unsigned char type, not as an int type.
-fconst_not_ROM	-fCNR	Does not handle the types specified by const as ROM data.
-fdouble_32	-fD32	This option specifies that the double type be handled in 32-bit data length as is the float type.
-fenable_register	-fER	Make register storage class available.
-fextend_to_int	-fETI	Performs operation after extending char-type data to the int type. (Extended according to ANSI standards.)[4]
-ffar_RAM	-fFRAM	Changes the default attribute of RAM data to far.
-finfo	None	Outputs the information required for the "Call Walker", "Map Function", and "utl30" to the absolute module file (.x30).
-fJSRW	None	Changes the default instruction for calling functions to JSR.W. When specify –OGJ,do not necessary to specify this option.
-fbit	-fB	Generates code assuming that bitwise manipulating instructions can be executed using absolute addressing for all external variables mapped into the near area.
-fno_carry	-fno_carry	Suppresses carry flag addition when data is indirectly accessed using far-type pointers.
-fauto_128	-fA1	Limits the usable stack frame to 128 byte.
-ffar_pointer	-fFP	Change the default attribute of pointer-type variable to far.
-fnear_ROM	-fNROM	Change the default attribute of ROM data to near.
-fno_align	-fNA	Does not align the start address of the function.
-fno_even	-fNE	Allocate all data to the odd section, with no separating odd data from even data when outputting.
-fno_switch_table	-fNST	When this option is specified, the code which branches since it compares is generated to a switch statement.
-fnot_address_volatile	-fNAV	Does not regard the variables specified by #pragma ADDRESS (#pragma EQU) as those specified by volatile.
-fnot_reserve_asm	-fNRA	Exclude asm from reserved words. (Only _asm is valid.)
-fnot_reserve_far_and_near	-fNRFAN	Exclude far and near from reserved words. (Only _far and _near are valid.)
-fnot_reserve_inline	-fNRI	Exclude far and near from reserved words. (Only _inline is made a reserved word.)
-fsmall_array	-fSA	When referencing a far-type array whose total size is unknown when compiling, this option calculates subscripts in 16 bits assuming that the array's total size is within 64 Kbytes.

[4] char-type data or signed char-type data evaluated under ANSI rules is always extended to inttype data.
This is because operations on char types (c1=c2*2/c3; for example) would otherwise result in an overflow and failure to obtain the intended result.

Table A.8 Generated Code Modification Options (2)

Option	Short form	Function
-fswitch_other_section	-fSOS	This option outputs a ROM table for a 'switch' statement to some other section than a program section.
-fchange_bank_always	-fCBA	This option allows you to write multiple variables to an extended area.
-fauto_over_255	-fAO2	Changes the stack frame size per function that can be reserved to 64 Kbytes.
-fsizet_16	-fS16	Change the type definition size_t from type unsigned long to type unsigned int
-fptrdifft_16	-fP16	Change the type definition ptrdiff_t from type signed long to type signed int
-fuse_DIV	-fUD	This option changes generated code for divide operation.
-fuse_MUL	-fUM	This option changes generated code for multiplication operation.
-fSB_auto	-fSBA	Changes SB registers from one to another before generating SB relative, one function at a time.
-R8C	None	Generates code suitable for the R8C Family.
-R8CE	None	Generates code suitable for the R8C Family with 64-KB or larger ROM.

-fansi

Modify generated code

Function: Validates the following command line options:

-fnot_reserve_asm:	Removes asm from reserved words
-fnot_reserve_far_and_near:	Removes far and near from reserved words
-fnot_reserve_inline:	Removes inline from reserved words
-fextend_to_int:	

Supplement: When this option is specified, the compiler generates code in conformity with ANSI standards.

-fchar_enumerator -fCE

Modify generated code

Function: Processes enumerator types not as int types but as unsigned char types.

Notes: The type debug information does not include information on type sizes.
Therefore, if this option is specified, the enum type may not be referenced correctly in some debugger.

-fconst_not_ROM -fCNR

Modify generated code

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

Supplement: The const-specified data by default is located in the ROM area. Take a look at the example below.

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

In this case, the array "array" is located as ROM area. By specifying this option, you can locate the "array" in the RAM area.
You do not normally need to use this option.

-fdouble_32 -fD32

Modify generated code

Function: This option specifies that the double type be handled in 32-bit data length as is the float type.

Supplement:

(1) For this option to be used, a function prototype must always be expressly written. Without a prototype declaration, the compiler may not be able to generate the correct code.
(2) When you specify this option, the debug information of the type double is processed as the type float. So, the data of the type double is displayed as the type float on C watch window and global window of Debug tool.
(3) Mathematical functions are replaced with single-precision mathematical functions.

-fenable_register -fER

Register storage class

Function: Allocates variables with a specified register storage class to registers.

Supplement: When optimizing register assignments of auto variables, 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 register assignments in the program under the above situation.
When this option is specified, the following register-specified variables are forcibly assigned to registers:

- Integral type variable
- Pointer variable

Notes: Because register specification in some cases has an adverse effect that the efficiency decreases, be sure to verify the generated assembly language before using this specification.

-fextend_to_int -fETI

Modify generated code

Function: Extends char type or signed char type data to int type data to perform operation (extension as per ANSI rules).

Supplement: In ANSI standards, the char-type or singed char-type data is always extended into the int type when evaluated. This extension is provided to prevent a problem in char-type arithmetic operations, e.g., c1 = c2 * 2 / c3; that the char type overflows in the middle of operation, and that the result takes on an unexpected value. An example is shown below.

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

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

In this case, the char type overflows when calculating [c2 * 2], so that the correct result may not be obtained.
Specification of this option helps to obtain the correct result. The reason why extension into the int type is disabled by default is because it is conducive to increasing the ROM efficiency any further.

-ffar_RAM -fFRAM

Modify generated code

Function: Change 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 the RAM data to be located in other areas than the near area (64K bytes area).

Notes: This option cannot be used in conjunction with the "-R8C" or "-R8CE" option.

-finfo

Modify generated code

Function: Outputs the information required for the "Call Walker", "Map Function", and "utl30".

Supplement: When using "Call Walker", "Map Function", or "utl30", the absolute module file ".x30" output by this option is needed.

Notes: No check is made for the use of global variables in the asm function. For this reason, use of the asm function even in "utl30" is ignored.
-finfo includes -g.

-fJSRW

Modify generated code

Function: Changes the default instruction for calling functions to JSR.W.

Supplement: When calling a function that has been defined external to the source file, the "JSR.A" command is used by default. This option allows it to be changed to the "JSR.W" command. Change to the "JSR.W" command helps to compress the generated code size. This option is useful when the program is relatively small not exceeding 32K bytes in size or ROM compression is desired.

Notes: Conversely, if a function is called that is located 32K bytes or more forward or backward from the calling position, the "JSR.W" command causes an error when linking. This error can be avoided by a combined use with "#pragma JSRA".

-fbit -fB

Modify generated code

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

Supplement: If the near external variables subject to bit manipulations are located in the M16C Series ,R8C Family memory space 0000h through 1FFFh, specification of this option helps to increase the code efficiency generated by the compiler.
If 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 high 16 bits of far-type pointers (32-bit pointer), assuming that the data is not mapped across the 64K bytes boundary. As a result, the code will be more efficient.

Notes: When far-type pointers are used to indirectly access memory dynamically allocated using the malloc function, etc., or ROM data mapped to the far area, be sure that the data is not accessed spanning a 64K bytes boundary.
This option cannot used simultaneously with the "-R8C" or "-R8CE" option.

Precautions concerning the compiler option -fbit(-FB)

When you use the compiler option -fbit(-FB), pay attention to the following points.
The error in Supplement may be output as a warning.
"16-bits unsigned value is out of range"

-fauto_128 -fA1

Modify generated code

Function: Limits the usable stack frame to 128 bytes

-ffar_pointer -fFP

Changes generated code

Function: Change the default attribute of pointer-type variable to far.
This option sets the default pointer size to 32-bits.

Supplement:
(1) The pointer type variable in this compiler is a near attribute as a default attribute. This option is used when changing the default attribute of a pointer type variable into a far attribute.
(2) The pointer variable which described the near qualifier is not influenced of this option. It always becomes a near attribute.

Example)

char near *p; // It processes as a near pointer.

-fnear_ROM -fNROM

Modify generated code

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

Supplement: The ROM data (const-specified variables, etc.) are located in the far area by default. By specifying this option you can locate the ROM data in the near area.
You do not normally need to use this option.

-fno_align -fNA

Modify generated code

Function: Does not align the start address of the function.

-fno_even -fNE

Modify generated code

Function: When outputting data, does not separate odd and even data. That is, all data is mapped to the odd sections (data_NO, data_FO, data_INO, data_IFO, bss_NO, bss_FO, rom_NO, rom_FO).

Supplement: By default, the odd-size and the even-size data are output to separate sections.
Take a look at the example below.

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

In this case, variable "c" and variable "i" are output to separate sections. This is because the even-size variable "i" is located at an even address. This allows for fast access when accessing in 16-bit bus width.
Use this option only when you are using the compiler in 8-bit bus width and when you want to reduce the number of sections.

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: When this option is specified, the code which branches since it compares is generated to a switch statement.

Supplement: Only when code size becomes smaller when not specifying this option, the code which used the jump table is generated.

Notes: For such a large function whose code size is larger than 32 Kbytes, if code which contains a jump table for a switch statement is generated, the program may not be branched to an appropriate address.
In that case, be sure to specify this option.
Please note that when a code which cannot be branched properly because of not specifying this option is generated, the compiler, assembler and linkage editor do not output any warning or error message.

-fnot_address_volatile -fNAV

Modify generated code

Function: Does not handle the global variables specified by "#pragma ADDRESS" or "#pragma EQU" or the static variables declared outside a function as those that are specified by volatile.

Supplement: If I/O variables are optimized in the same way as for variables in RAM, the compiler may not operate as expected. This can be avoided by specifying volatile for the I/O variables.
Normally #pragma ADDRESS or #pragma EQU operates on I/O variables, so that even though volatile may not actually be specified, the compiler processes them assuming volatile is specified. This option suppresses such processing.

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

-fnot_reserve_asm -fNRA

Modify generated code

Function: Removes asm from the list of reserved words.

Supplement: "_asm" that has the same function is handled as a reserved word.

-fnot_reserve_far_and_near -fNRFAN

Modify generated code

Function: Removes far and near from list of reserved words.

Supplement: "_far" and "_near" that has the same function is handled as a reserved word.

-fnot_reserve_inline -fNRI

Modify generated code

Function: Does not handle inline as a reserved word.

Supplement: "_inline" that has the same function is handled as a reserved word.

-fsmall_array -fSA

Modify generated code

Function: When referencing a far-type array whose total size is unknown when compiling, this option calculates subscripts in 16 bits assuming that the array's total size is within 64K bytes.

Supplement: If when referencing array elements in a far-type array such as array data in ROM, the total size of the far-type array is uncertain, the compiler calculates subscripts in 32 bits in order that arrays of 64K bytes or more in size can be handled.
Take a look at the example below.

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

In this case, because the total size of the array array is not known to the compiler, the subscript "j" is calculated in 32 bits.
When this option is specified, the compiler assumes the total size of the array array is 64 K bytes or less and calculates the subscript "j" in 16 bits. As a result, the processing speed can be increased and code size can be reduced.
Renesas recommends using this option whenever the size of one array does not exceed 64K bytes.

-fswitch_other_section -fSOS

Modify generated code

Function: This option outputs a ROM table for a 'switch' statement to some other section than a program section.

Supplement: Section name is 'switch_table'

Notes: This option does not normally need to be used.

-fchange_bank_always -fCBA

Modify generated code

Function: This option allows you to write multiple variables to an extended area.(with #pragma EXT4MPTR or _ext4mptr)

Supplement: Specify this option when you declare multiple pointer variables to a 4M bytes space while at the same time using the #pragma EXT4MPTR or _ext4mptr feature.

Notes: This option cannot be used in conjunction with the "-R8C" or "-R8CE" option.

-fauto_over_255 -fAO2

Modify generated code

Function: Changes the stack frame size per function that can be reserved to 64K bytes.
(The maximum value in the default of the stack frame is 255 bytes.)

Notes:

1. This option cannot be used in combination with #pragma SBDATA. If a file that contains a description of #pragma SBDATA is compiled, the warning shown below is output, with the description of #pragma SBDATA ignored.

   [Warning(ccom):XX.c,line XX] compile option –fauto_over_255 is specified, #pragma SBDATA was ignored.
   ==> #pragma SBDATA xxx;

2. Specify this option for the files described below.
   a. When a function exists that requires a stack frame of 255 bytes or more

-fsizet_16 -fS16

Change the bit size of type definition

Function: Change the type definition size_t from type unsigned long to type unsigned int

Notes: If this option is selected,be sure to use one of the standard function libraries listed below when linking.

- M16C/60series
  nc30s16.lib
- R8C Family
  r8cs16.lib

-fptrdifft_16 -fP16

Change the bit size of type definition

Function: Change the type definition ptrdiff_t from type signed long to type signed int

Notes: If this option is selected, be sure to use one of the standard function libraries listed below when linking

- M16C/60 series
  nc30s16.lib
- R8C Family
  r8cs16.lib

-fuse_DIV -fUD

Modify generated code

Function: This option changes generated code for divide operation.

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 div.w (divu.w) and div.b (divu.b) microcomputer instructions.

Notes:

(1) If the divide operation results in an overflow when this option is specified, the compiler may operate differently than stipulated in ANSI.

(2) The div instruction of the M16C Series or R8C Family has such a characteristic that when the operation resulted in an overflow, the result becomes indeterminate. Therefore, when the program is compiled in default settings by NC30, it calls a runtime library to correct the result for 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: This option changes generated code for multiplication operation.

Supplement: When 16 bits×16 bits is stored in 32 bits, it should be Cast in 32 bits of the multiplier or the multiplicand because it obtains the result of high rank 16 bits.
The result of 32bit can be obtained by specifying the option Cast.

-R8C

Modify generated code

Function: Generates code suitable for the R8C Family.

Supplement: The _fnear_ROM (-fNROM) option is set by default.

Notes: This option cannot be used in combination with the following options.
If one of these options is specified, the option is ignored.
-ffar_RAM(- fFRAM), -fno_carry(- fNC), -fchange_bank_always(- fCBA)

-R8CE

Modify generated code

Function: Generates code suitable for the R8C Family.

Notes:

(1) This option cannot be used in combination with the options listed below. If one of these options is specified, the option is ignored.
-ffar_RAM(-fFRAM)
-fno_carry(-fNC)
-fchange_bank_always(- fCBA)

(2) When ROM area exceeds 64K boundary, it uses it.

-fSB_auto -fSBA

Modify generated code

Function: Changes SB registers from one to another before generating SB relative, one function at a time.

Supplement: Analyzes the number of times external variables are referenced in a function 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 address of _sym is made
        sym = a * b;     the base point for SB relative.
        if(sym != 0)
                sym = sub();
        return sym;
}
int data1,data2;
int sub(void)
{                        The address of _data1 is
        data1 = sym1;    made the base point for SB
        data2 = data1/2; relative.
        data1 = sub1(data2);
                :
                :
}
```

(1) The address of the symbol that was made the base point for SB relative is stored in the SB register.
(2) At the entry and exit to and from the function, code is generated for saving/restoring the SB register.
(3) Only external variables are effective.
(4) This option cannot be used in combination with -OR, -OS, -OR_MAX, -ORM, -OS_MAX and -OSM.

A.2.7 Library Specifying Option

Table A.9 lists the startup options you can use to specify a library file.

Table A.9 Library Specifying Option

Option	Function
-l*libraryfilename*	Specifies a library file that is used by ln30 when linking files.

-l*library-file-name*

Specifying a library file

Function: Specifies a library file that is used by ln30 when linking files. The file extension can be omitted.

Syntax: nc30Δ-l*filename*Δ<*C source file name*>

Notes:

(1) In file specification, the extension can be omitted. If the extension of a file is omitted, it is processed assuming an extension ".lib".

(2) If you specify a file extension, be sure to specify ".lib".

(3) NC30 links by default a library "nc30lib.lib" in the directory that is specified in environment variable LIB30. (NC30 links "r8clib.lib" when compiler option "-R8C" has been specified.)

(4) If multiple libraries are specified, references to “nc30lib.lib” are assigned the lowest priority.

A.2.8 Warning Options

Table A.10 shows the command line options for outputting warning messages for contraventions of nc30 language specifications.

Table A.10 Warning Options

Option	Short form	Function
-Wall	None	Displays message for all detectable warnings. (however, not including alarms output by -Wlarge_to_small and "-Wno_used_argument")
-Wccom_max_warnings =*Warning Count*	-WCMW	This option allows you to specify an upper limit for the number of warnings output by ccom30.
-Werror_file<*file name*>	-WEF	Outputs error messages to the specified file.
-Wlarge_to_small	-WLTS	Outputs a warning about the tacit transfer of variables in descending sequence of size.
-Wmake_tagfile	-WMT	Outputs error messages to the tag file of source file by source file.
-Wnesting_comment	-WNC	Outputs a warning for a comment including "*/" .
-Wno_stop	-WNS	Prevents the compiler stopping when an error occurs.
-Wno_used_argument	-WNUA	Outputs a warning for unused argument of functions.
-Wno_used_function	-WNUF	Displays unused global functions when linking.
-Wno_used_static_function	-WNUSF	For one of the following reasons, a static function name is output that does not require code generation.
-Wno_warning_stdlib	-WNWS	Specifying this option while "-Wnon_prototype" or "-Wall" is specified inhibits "Alarm for standard libraries which do not have prototype declaration.
-Wnon_prototype	-WNP	Outputs warning messages for functions without prototype declarations.
-Wstdout	None	Outputs error messages to the host machine's standard output (stdout).
-Wstop_at_link	-WSAL	Stops linking the source files if a warning occurs during linking to suppress generation of absolute module files. Also, a return value "10" is returned to the host OS.
-Wstop_at_warning	-WSAW	Stops compiling the source files if a warning occurs during compiling and returns the compiler end code "10".
-Wundefined_macro	-WUM	Warns you that undefined macros are used in #if.
-Wuninitialize_variable	-WUV	Outputs a warning about auto variables that have not been initialized.
-Wunknown_pragma	-WUP	Outputs warning messages for non-supported #pragma.

-Wall

Warning Options

Function: Indicates all detectable alarms.

Supplement:

(1) The alarms indicated here do not include those that may be generated when "Wlarge_to_small(-WLTS)" and "Wno_used_argument(-WNUA)" and "Wno_used_static_function(-WNUSF)" are used.

(2) The alarms indicated here are equivalent to those of the options "Wnon_prototype(-WNP)," "Wunknown_pragma(-WUP)," "Wnesting_comment(-WNC)," and "Wuninitialize_variable(-WUV)."

(3) Alarms are indicated in the following cases too:

- When the assignment operator = is used in the if statement, the for statement or a comparison statement with the && or | | operator.
- When "= =" is written to which '=' should be specified.
- When function is defined in old format.

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

-Wccom_max_warnings= *Warning Count* -WCMW= *Warning Count*

Warning Options

Function: This option allows you to specify an upper limit for the number of warnings output by ccom30.

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 you specify 0, warning outputs are completely suppressed inhibited.

-Werror_file <*file-name*>

Warning Options

Function: Outputs error messages to the specified file.

Syntax: nc30Δ-W*error_file*Δ<*output error message file name*>

Notes: The format in which error messages are output to a file differs from one in which error messages are displayed on the screen. When error messages are output to a file, they are output in the format suitable for the "tag jump function" that some editors have.

-Wlarge_to_small -WLTS

Warning Options

Function: Outputs a warning about the substitution of variables in descending sequence of size.

Supplement: A warning may be output for negative boundary values of any type even when they fit in the type. This is because negative values 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 value 32768 does not fit in the signed int type and, consequently, becomes the signed long type.
Therefore, the immediate value 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 immediate values to char type variables
- Assignment of immediate values to float type variables

-Wmake_tagfile -WMT

Warning Options

Function: Outputs error messages to the tag file of source-file by source-file, when an error or warning occurs.

Supplement: This option with "-Werror_file (-WEF)" option can't specify.

-Wnesting_comment -WNC

Warning Options

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

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

Precautions concerning the compiler option -Wlarge_to_small(-WLTS)

When you use the compiler option -Wlarge_to_small(-WLTS), pay attention to the following.
When compiled as a C program, a warning is output when the right side consists only of a variable.

-Wno_stop -WNS

Warning Options

Function: Prevents the compiler 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, another error may be induced by an error, 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.

-Wno_used_argument -WNUA

Warning Options

Function: Outputs a warning for unused arguments function.

-Wno_used_function -WNUF

Warning Options

Function: Displays unused global functions when linking.

Notes: When selecting this option, be sure to specify the "-finfo" option at the same time.
When -U option is specified when linking, this option is unnecessary.

-Wno_used_static_function -WNUSF

Warning Options

Function: For one of the following reasons, a static function name is output that does not require code generation.

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

Notes: (1) If any function name is written in an array initialize in the manner 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])();


* In the above example, although functions f4 and f5 are not referenced, the compiler processes
  these functions assuming that they will be referenced.
```

-Wno_warning_stdlib -WNWS

Warning Options

Function: Specifying this option while "-Wnon_prototype" or "-Wall" is specified inhibits "Alarm for standard libraries which do not have prototype declarations".

-Wnon_prototype -WNP

Warning Options

Function: Outputs warning messages for functions without prototype declarations or if the prototype declaration is not performed for any function.

Supplement: Function arguments can be passed via a register by writing a prototype declaration. Increased speed and reduced code size can be expected by passing arguments via a register. Also, the prototype declaration causes the compiler to check function arguments. Increased program reliability can be expected from this.
Therefore, Renesas recommends using this option whenever possible.

-Wstdout

Warning Options

Function: Outputs error messages to the host machine's standard output (stdout).

Supplement: Use this option to save error output, etc. to a file by using Redirect.

Notes: In this Compiler, errors from assembler and linkage editor invoked by the compile-driver are output to the standard output regardless of this option.

-Wstop_at_link -WSAL

Warning Options

Function: Stops linking the source files if a warning occurs during linking to suppress generation of absolute module files. Also, a return value "10" is returned to the host OS.

-Wstop_at_warning -WSAW

Warning Options

Function: Stops compiling the source files if a warning occurs during compiling and returns the compiler end code "10."

Supplement: If a warning occurs when compiling, the compilation by default is terminated with the end code "0" (terminated normally).
Use this option when you are using the make utility, etc. and want to stop compile processing when a warning occurs.

-Wundefined_macro -WUM

Warning Options

Function: Warns you that undefined macros are used in #if.

-Wuninitialize_variable -WUV

Warning Options

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 or a for statement in the user application, the compiler assumes it is not initialized.
Therefore, when this option is used, the compiler outputs a warning for it.

-Wunknown_pragma -WUP

Warning Options

Function: Outputs warning messages for non-supported #pragma.

Supplement: By default, no alarm is generated even when an unsupported, unknown "#pragma" is used.
When you are using only the NC-series compilers, use of 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

Table A.11 shows the command line options for specifying as30 and ln30 options.

Table A.11 Assemble and Link Options

Option	Function
-as30Δ< Option>	Specifies options for the as30 link command. If you specify two or more options, enclose them in double quotes.
-ln30Δ< Option>	Specifies options for the ln30 assemble command. If you specify two or more options, enclose them in double quotes.

-as30 "*Option*"

Assemble/link option

Function: Specifies as30 assemble command options
If you specify two or more options, enclose them in double quotes.

Syntax: nc30Δ-as30Δ"*option1Δoption2*"Δ<*C source file*>

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

-ln30 "*Option*"

Assemble/link option

Function: Specifies options for the ln30 link command. You can specify a maximum of four options.
If you specify two or more options, enclose them in double quotes.

Syntax: nc30Δ-ln30Δ"*option1Δoption2*"Δ<*C source file name*>

Notes: Do not specify the ln30 options "-.", "-G", "-O", "-ORDER", "-L", "-T", "-V" or "@ file".

A.3 Notes on Command Line Options

A.3.1 Coding Command Line Options

The NC30 command line options differ according to whether they are written in uppercase or lowercase letters. Some options will not work if they are specified in the wrong case.

A.3.2 Priority of Options for Controlling

If you specify both the following options in the NC30 command line, the -S option takes precedence and only the assembly language source files will be generated.

- "-c":Stop after creating relocatable module files.
- "-S":Stop after creating assembly language source files.

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.

Table B.1 Extended Functions (1)

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 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)

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.

Inline storage class	Functions can be inline developed by using the inline storage class specifier.inline. Example: inline func(int i); * Always be sure to define the body of an inline function before using the inline function.
Extension of Comments	You can include C++-like comments ("//"). Example: // This is a comment.
#pragma Extended functions	You can use extended functions for which the hardware of M16C Series, R8C Family in C language.

macro assebler function	You can describe some assembler command as the function of C
	Example:
	char dadd_b(char val1, char val2);
	Example:
	int dadd_w(char val1, char val2);

B.1 Near and far Modifiers

For the M16C/60 series 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.

* near modifier................................Area of 000000H to 00FFFFH
* 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:

* Variables ....near attribute
* const-qualified constants........... far attribute
* 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

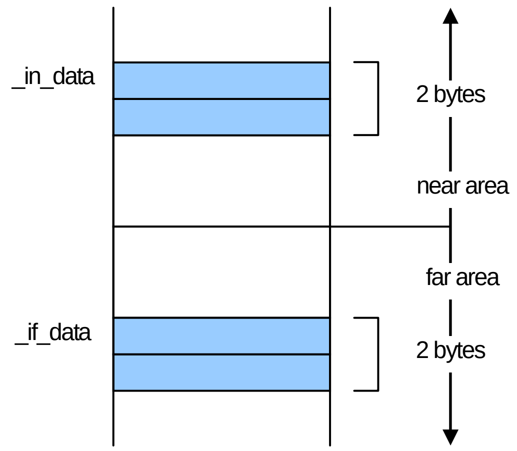


Figure B.3 Memory Location of Variable

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 Declarning 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 shows memory maps for abobe example.

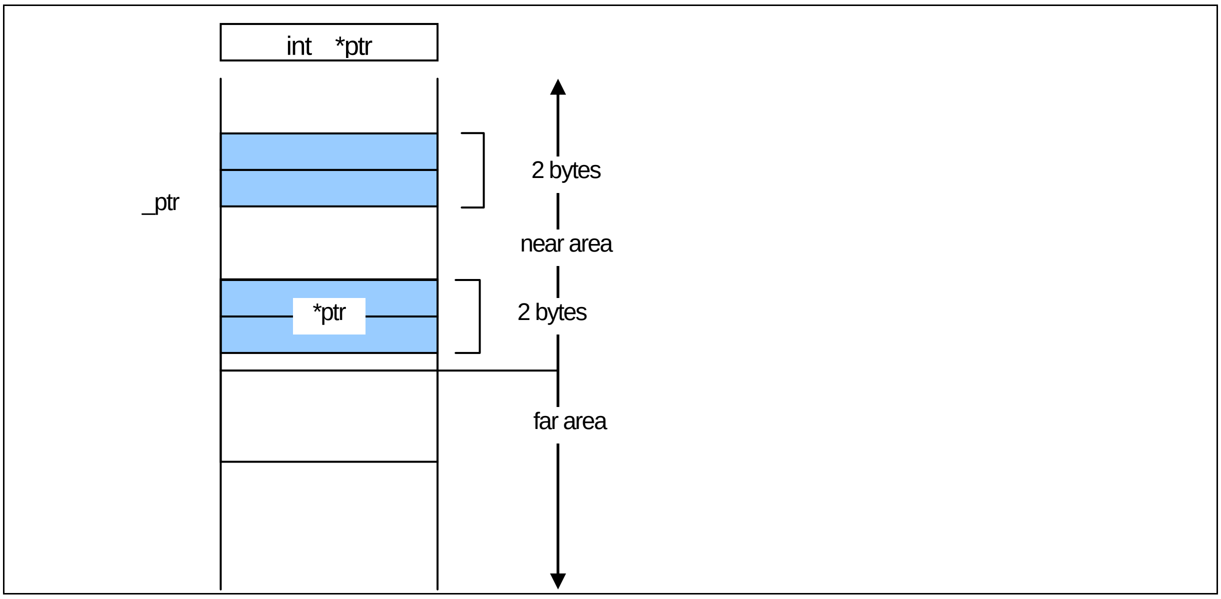


Figure B.6 Memory Location of Pointer type Variable

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 (1)

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 (2)

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.

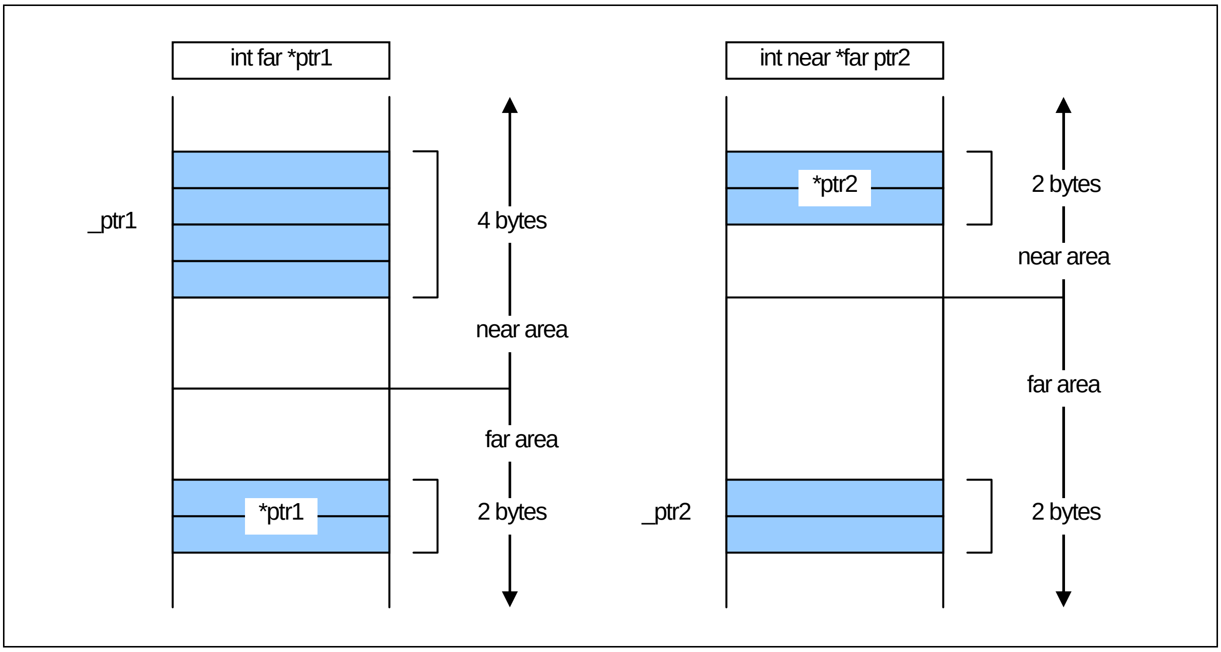


Figure B.9 Memory Location of Pointer type Variable

B.1.4 Format of Function Declaration

A function's near and far allocation attributes are always far. If you specify the near attribute in function declaration, the system outputs a warning message (function must be far) with your near declaration ignored.

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

NC30 handles the attribute of far and the variable as near with the attribute of the function if you do not specify the near and far attributes. NC30's command line options allow you to modify the default attributes of functions and variables (data). These are listed in the table below.

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.

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 );         /* prototype declaration for function with far pointer to parameter */
                                    /* specifies near pointer parameter at the function call */
}
```

Figure B.10 Type conversion from near to far

When converting type into far, 0 (zero) is expanded as high-order address.

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

When compiling, the warning message "assign far pointer to near pointer, bank value ignored" is output for the code shown in Figure B.11 to show that the high part of the address (the bank value) has been 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_pyr );         /* prototype declaration of function */
                                    /* with near pointer in 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.

B.1.8 Declaring functions

In NC30, functions are always located in the far area. Therefore, do not write a near declaration for functions.
If a function is declared to take on a near attribute, NC30 outputss a warning and contin¬ues processing by assuming the attribute of that function is far. Figure B.12 shows a dis¬play example where a function is declared to be near.

```
%nc30   -S   smp.c
M16C Series,R8C Family NC30 COMPILER V.X.XX Release XX
Copyright(C) XXXX(XXXX-XXXX). Renesas Electronics Corporation.
and Renesas Solutions Corp., All rights reserved.
smp.c
[Warning(ccom):smp.c,line 3] function must be far
===> {
func
%
```

Figure B.12 Example Declaration of Function

B.1.9 Function for Specifying near and far in Multiple Declarations

As shown in Figure B.13, if there are multiple declarations of the same variable, the type information for the variable is interpreted as indicating a 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

B.1.10 Notes on near and far Attributes

a. Notes on near and far Attributes of Functions

Functions always assume the far attribute. Do not declare functions with near. NC30 will output a warning when you declare the near attribute for a function.

b. 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

B.2 asm Function

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

B.2.1 Overview of asm Function

The asm function is used for including assembly language code in a C 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:

- Specifying the FB offset of storage class auto variables in the C program using the names of the variables in C
- Specifying the register name of storage class register variables in the C program using the names of the variables in C
- Specifying the symbol name of storage class extern and static variables in the C program using the names of the variables in C

The following shows precautions to be observed when using the asm function.

- Do not change the contents of registers within an asm function.
  The compiler does not check the inside of the asm function. If registers are going to be destroyed, write push and pop instructions using the asm function to save and restore the registers.

[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 storage class auto and register variables (including arguments) written in the C language are referenced and located as being offset from the Frame Base Register (FB). (They may be mapped to registers as a result of optimization.)
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( "    op-code    R1 , $$ [ 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)
                  :
}
```

- Assembly language source 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 og greater than 2-bits.)

```
asm( "      op-code       $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);
}
```

- Assembly language 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'
            .align
            ._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

When referencing a bit field in the auto area,you must confirm that it is located within the range that can be referenced using bit operation instructions(within 32 bytes of the FB register value).

B.2.3 Specifying Register Name of register Variable

The storage class auto and register variables (including arguments) may be mapped 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( "      op-code    operand $$ " , 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;        ← Variable" i" is a register variable

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

- Assembly language source file (compile result):

```
;## # FUNCTION func
;## # ARG Size(0) Auto Size(0) Context Size(4)
            .section    program,CODE,ALIGN
            ._file      'reg.c'
            .align
            ._line 3
;## # C_SRC : {
            .glb        _func
_func:
            ._line 4
;## # C_SRC : register 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

extern and static storage class variables written in C are referenced as symbols.
You can use the format shown in Figure B.26 to use extern and static variables in asm functions.

```
asm( "     op-code     operand, $ " , variable name );
```

Figure B.26 Description Format for Specifying Symbol Name

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)

```
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;      ← static variable

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

- Assembly language source file(compile result):

```
_func:
             ._line 7
;## # C_SRC : asm("   mov.w      #01H,$$",ext_val);
;#### ASM START
             mov.w     #01H,_ext_val          ← Move to_ext_val
             ._line 8
;## # C_SRC : asm(" mov.w         #01H,$$",s_val);
             mov.w     #01H,___S0_s_val     ← Move to__S0_e_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 og greater than 2-bits.)

```
asm( "     op-code      $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:
          .section  bss_NO,DATA
          .glb      _s
_s:
          .blkb 1
          .END
```

Figure B.31 Example of Referencing Bit Field of Symbol

When referencing the bit fields of extern or static variables, you must confirm that they are located within the range that can be referenced directly using bit operation instructions (within 0000H and 1FFFH).

B.2.5 Specification Not Dependent on Storage Class

The variables written in C language can be used in the asm function without relying on the storage class of that variable (auto, register[1], extern, or static variable).
Consequently, any variable written in C language can be used in the asm function by writing it in the format shown in Figure B.32[2]

```
asm("    op-code operand, $@", variable name );
```

Figure B.32 Description Format Not Dependent on Variable's Storage Class

You can only specify two variable 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;       ←extern variable

void          func(void)
{
                  int        f_val; .    ← auto variable
              register int   r_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);
}
```

- Assembly language 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 Selectively suppressing optimization

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;
}
```

Optimization results in any steps to set the two port bits separately being combined as one step.

Optimization ▶

```
or.b #03H,_port
```

```
          port.bit0 = 0x01;
          asm();      /*dummy*/
          port.bit1 = 0x01;
```

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) Do not specify auto variables or parameters, or 1-bit bit fields using the offset from the frame base register (FB). Use the format shown in Figure B.35 to specify auto variables and parameters.

```
asm("   MOV.W   #01H,$$[FB]", i);     ← Format for referencing auto variables
asm("   BSET    $$[FB]", s.bit0);     ← Format for checking auto bit fields
```

Figure B.35 Example Coding of asm Function (1)

(2) 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)

Note that, when you specify option -O[1-5], -OR, -OS, -OR_MAX, or -OS_MAX, parameters passed via the registers may, to improve code efficiency, be processed as register variables rather than being moved to the auto area. 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

- In assembly-language descriptions, do not write statements which will cause the register contents to be changed. If registers are going to be destroyed, use push and pop instructions to save and 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 inFigure 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
Examples:  A1:
           C9830:


● Labels consisting of two or more characters preceded by the underscore (_)
Examples:  __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)

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:

set NCKIN=SJIS
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.

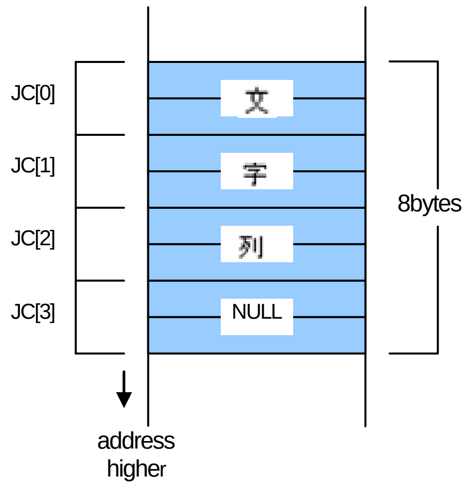


Figure B.43 Memory Location of wchar_t Type Character Strings

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.

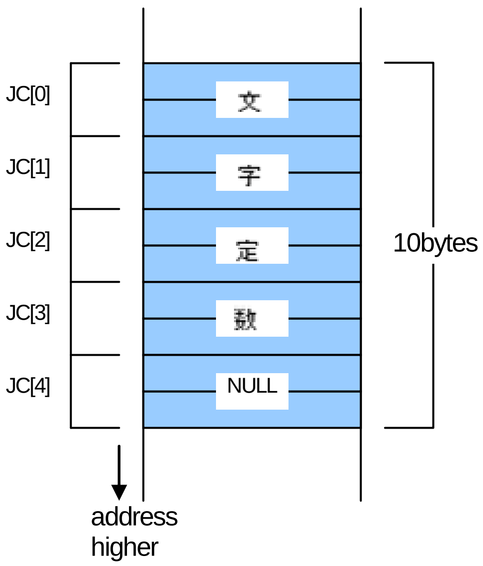


Figure B.46 Memory Location of wchar_t Type Character Constant Assigned Array

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.

B.4.1 Overview of Default Argument Declaration of Function

NC30 allows you to use implicit arguments by assigning parameter default values when declaring a function's prototype. By using this facility you can save the time and labor that would otherwise be required for writing frequently used values when calling a function.

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

Figure B.48 shows an example of declaration of a function, and Figure B.49 shows a result of compiling of sample program which shows atFigure B.48.

```
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.Figure B.50 shows an example where default arguments are specified with variables. Figure B.51 shows a compile result of the sample program shown inFigure B.50.

```
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 Examples of Prototype Declaration

b. When specifying a variable for a default value

When specifying a variable for a default value, write the prototype declaration of a function after declaring the variable you specify. If a variable is specified for the default value of an argument that is not declared before the prototype declaration of a function, it is processes 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 chapter describes specifications of the inline storage class.

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. type specifier. function;
```

Figure B.53 Declaration Format of inline Storage Class

Figure B.54 shows an example of declaration of a function.

```
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 Example for Declaring inline Storage Class

```
          .SECTION  program,CODE,ALIGN
          ._file    'sample.c'
          .align
          ._line    7
;## # C_SRC :        {
          .glb      _main
_main:
          enter     #02H
          pushm     R1
          ._line    10
;## # C_SRC :                  s = func(s);
          mov.w     -2[FB],R1   ;  s
          ._line    3
;## # C_SRC :                  return i++;          ← 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:
          .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 parameter of inline functions

The parameter of an in line function cannot be used by “structure” and “union”.It becomes a compile error.

(2) 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.

(3) 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.

(4) 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]
[Error(ccom):sample.c,line 5] inline function's body is not declared previously
===> func(1);
Sorry, compilation terminated because of these errors in main().

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]
[Error(ccom):in.c,line 9] inline function is called as normal function before
===>{

Figure B.57 Example of inappropriate code of inline function (2)

(5) 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]
[Error(ccom):sample.c,line 10] can't get inline function's address by '&' operator
===> f = &func;
Sorry, compilation terminated because of these errors in main().

Figure B.58 Example of inappropriate code of inline function (3)

(6) 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]
[Warning(ccom):smp.c,line 3] static valuable in inline function
===> static int i = 0;

Figure B.59 Example of inappropriate code of inline function (4)

(7) 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 Extension of Comments

NC30 allows comments enclosed between "/*" and "*/" as well as C++-like comments starting with "//".

B.6.1 Overview of "//" Comments

In C, comments must be written between "/*" and "*/". In C++, anything following "//"

B.6.2 Comment "//" Format

When you include "//" on a line, anything after the "//" is treated as a comment.
Figure B.60 shows comment format.

```
// comments
```

Figure B.60 Comment Format

Figure B.61 shows example comments.

```
void        func(void)
{
            int i;              /* This is commentes *//
            int j;              // This is commentes
               :
            (omitted)
               :
}
```

Figure B.61 Example Comments

B.6.3 Priority of "//" and "/*"

The priority of "//" and "/*" is such that the one that appears first has priority.
Therefore, a "/*" written between a "//" to the new-line code does not have an effect as signifying the beginning of a comment. Also, a "//" written between "/*" and "*/" does not have an effect as signifying the beginning of a comment.

B.7 #pragma Extended Functions

B.7.1 Index of #pragma Extended Functions

Following index tables show contents and formation for #pragma[1] extended functions.

a. Using Memory Mapping Extended Functions

Table B.4 Memory Mapping Extended Functions

Extented function	Description
#pragma ROM	Maps the specified variable to rom Syntax : #pragma ROM variable_name Example : #pragma ROM val ● This facility is provided to maintain compatibility with NC77 and NC79. ● The variable normally must be located in the rom section using the const qualifier.
#pragma BIT	Declares that the external variable resides in an area where a 1-bit manipulate instruction can be used in 16-bit absolute ad-dressing mode (i.e., a variable residing in addresses from 00000H to 01FFFH). Syntax : #pragma BIT *variable name* Example : #pragma BIT bit_data
#pragma SBDATA	Declares that the data uses SB relative addressing. Syntax : #pragma SBDATA variable name Example : #pragma SECTION bss nonval_data
#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
#pragma EXT4MPTR	A functional extension which shows a variable is a pointer accessing 4-Mbyte expanded space ROM. Syntax : #pragma EXT4MPTR 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 variable name Example : _ext4mptr far *sym

[1] In the previous versions, words following #pragma (For example, ADDRESS, INTERRUPT, ASM ,etc.)specifying a directive function (abbreviate as subcommand) needed to be described in uppercase. Inthis version, subcommand are case-independence, in which uppercase and lowercase are considered to be equivalent.

b. Using Extended Functions for Target Devices

Table B.5 Extended Functions for Use with Target Devices (1)

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 written in assembler called in a software interrupt (int instruction). Syntax：#pragma INTCALL INT-No.. function-name(registe-name) Example：#pragma INTCALL 25 func(R0, R1) Example：#pragma INTCALL 25 func(R0, R1) Syntax：#pragma INTCALL INT-No. function-name() Example：#pragma INTCALL 25 func() Example：#pragma INTCALL 25 func() ● Always be sure to declare the prototype of the function before entering this declaration.
#pragma INTERRUPT	Declares an interrupt handling function written in C language. This declaration causes code to perform a procedure for the interrupt handling function to be generated at the entry or exit to and from the function. Furthermore, by specifying switch /B it is possible to switch the register to a back register instead of saving it to a stack when calling the function. Syntax： #pragma INTERRUPT [/B|/E|/V] interrupt-handling-function-name #pragma INTERRUPT [/B|/E] interrupt-vector-number. interrupt-handlingfunction-name 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) #pragma INTERRUPT /V int_func ()

Table B.6 Extended Functions for Use with Target Devices (2)

Extended function	Description
#pragma PARAMETER	Declares that, when calling an assembler function, the parameters are passed via specified registers. Syntax : #pragma PARAMETER function_name (register_name) Example : #pragma PARAMETER asm_func(R0,R1) Example : #pragma PARAMETER asm_func(R0,R1) ● Always be sure to declare the prototype of the function before entering this declaration.
#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)

c. The Other Extensions

Table B.7 Using Inline Assembler Description Function

Extended feature	Description
#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 JSRA	Calls functions using JSR.A as the JSR instruction. Syntax : #pragma JSRA function-name Example : #pragma JSRA func
#pragma JSRW	Calls functions using JSR.W as the JSR instruction. Syntax : #pragma JSRW function-name Example : #pragma JSRW func
#pragma PAGE	Indicates a new-page point in the assembler listing file. Syntax : #pragma PAGE Example : #pragma PAGE
#pragma __ASMMACRO	Declares defined a function by assembler macro. Syntax : #pragma __ASMMACRO. function-name(register name, ...) Example : #pragma __ASMMACRO mul(R0,R1)

B.7.2 Using Memory Mapping Extended Functions

NC30 includes the following memory mapping extended functions.

#pragma ROM

Map to rom section

Function: Maps specified data (variable) to rom section

Syntax: #pragma ROM. variable_name

Description: This extended function is valid only for variables that satisfy one or other of the following conditions:

- Non-extern variables defined outside a function (Variables for which an area is secured)
- Variables declared as static within the function

Rules:
(1) If you specify other than a variable, it will be ignored.
(2) No error occurs if you specify #pragma ROM more than once.
(3) The data is mapped to a rom section with initial value 0 if you do not include aninitialization expression.

Example:

```
[C language source program]

#pragma   ROM   i
unsigned int          i;                 ← Variable i, which satisfies condition[1]

void         func(void)
{
             static int    i = 20;       ← Variable i, which satisfies condition[2]
                 :
             (remainder omitted)


[Assembly language source program]

             .SECTION  rom_NE,ROMDATA
___S0_i: ;### C's name is i              ← Variable i, which satisfies condition[2]
             .word         0014H
             .glb          _i
_i:                                      ← Variable i, which satisfies condition[1]
             .byte         00H
             .byte         00H
```

Figure B.62 Example Use of #pragma ROM Declaration

Note: This facility is provided to maintain compatibility with NC77 and NC79. The variable normally must be located in the rom section using the const modifier.

#pragma BIT

SB Relative Addressing Using Variable Description Function

Function: Declares an external variable that exists in an area where a one-bit manipulate instruc-tion can be used in 16-bit absolute addressing mode.

Syntax: #pragma BIT *variable_name*

Description: The M16C series,R8C Family allows you to use a one-bit manipulate instruction for external variables located in an area of addresses 00000H to 01FFFH in a ROM efficient, 16-bit absolute addressing mode.
The variable declared by #pragma BIT is assumed to be present in an area where a one-bit manipulate instruction can be operated on it directly.

Rules:
(1) If #pragma BIT is used for anything other than an external variable, it is ignored as invalid.
(2) When an external variable is declared in #pragma BIT and also has a bit width of 1 bit, always directly output 1-bit instructions.
It is therefore the user's responsibility to ensure that, when #pragma BIT declarations are included, the variables are mapped between 0 and 01FFFH.

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;
```

Figure B.63 Example Use of #pragma BIT Declaration

Note: 1-bit instructions in a 16-bit absolute addressing mode are generated under the following either conditions:
(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: The M16C series,R8C Family allows you to choose instructions that can be executed efficiently by using SB relative addressing. #pragma SBDATA declares that SB relative addressing can be used for the variable when referencing data. This facility helps to generate ROM-efficient code.

Rules:

(1) The variable declared to be #pragma SBDATA is declared by the assembler's pseudo-instruction .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[1]

(6) Declaration of #pragma SBDATA is invalid when the –fauto_over_255 (-fAO2) option is specified. In such a case, NC30 outputs a warning message “compile option -fauto_over_255 is specified, #pragma SBDATA was ignored”.

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.64 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.

[1] Do not write a #pragma SBDATA declaration for ROM data.

#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.

- You can specify "#pragma SECTION bss", "#pragma SECTION rom", "#pragma SECTION data" and "#pragma SECTION program" two or more times in one file.
- All other sections cannot have their names changed twice or more.

Example :

```
[C source program]

#pragma   SECTION   program   pro1      ← Changes name of program section to pro1
void      func( void );
               :
          (remainder omitted)


[Assembly language 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.65 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.

#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, bss, and interrupt 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

Description and Examples :

In NC30, structures are packed by default. For example, the size of the structure in Figure B.66 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.66 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. Figure B.67 shows an example in which #pragma STRUCT is used to inhibit packing of the structure in Figure B.66.

```
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.67 Example Mapping of Structure Members (2)

As shown Figure B.67, 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

Description : (2) Arranging members

This NC30 extended function allows you to map the all even-sized structure members first, followed by odd-sized members. Figure B.68 shows the offsets when the structure shown in Figure B.66 is arranged using #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.68 Example Mapping of Structure Members (3)

You must declare #pragma STRUCT for inhibiting packing and arranging the structure members before defining the structure members.

Examples :

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

Figure B.69 Example of #pragma STRUCT Declaration

#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.

Examples :

```
[C source program]
struct tagh{
        int bitmap;
        char code;
}far *pointer;
#pragma EXT4MPTR pointer
main()
{
        int data;
        data = pointer->bitmap;
}
        mov.w _pointer, A0
        mov.w _pointer+2, A1
        mov.w A1,__BankSelect    ← Change the bank
        bclr 3,A1
        bset 2,A1
        lde.w [A1A0],-2[FB]
```

Figure B.70 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 or –R8CE is used, this declaration is ignored.

_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_name

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.

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.71 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 or –R8CE is used, this declaration is ignored.

B.7.3 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 ADDRESSΔvariable-nameΔabsolute-address

Description : The absolute address specified in this declaration is expanded as a character string in an assembler file and defined in pseudo instruction .EQU. The format for writing the numerical values therefore depends on the assembler, as follows:

- 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) #pragma ADDRESS is valid for previously declared variables.
(4) #pragma ADDRESS is invalid if you specify other than a variable.
(5) 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) Normally #pragma ADDRESS operates on I/O variables, so that even though volatile may not actually be specified, the compiler processes them assuming volatile is specified.
(8) The variable declared in #pragma ADDRESS declaration, external reference is impossible.
(9) When the -fnot_address_volatile(-fNAV) option is specified, the volatile attribute for the variable defined with #pragma ADDRESS becomes invalid.
(10) If a string that follows "#pragma ADDRESS" consists of white spaces or characters that cannot be a variable name (e.g. 123), or a specified address only consists of white spaces, NC30 outputs a warning message "#pragma ADDRESS format error,ignored" and ignores this statement.
(11) If a string entered as the address contains a character whose 8th bit is 1, NC30 outputs a warning message "Kanji in #pragma ADDRESS" and ignores this statement.

Examples :

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

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

Figure B.72 #pragma ADDRESS Declaration

#pragma ADDRESS

Specify absolute address of I/O variable

Examples : However, as follows, when the variable is used before specification of #pragma ADDRESS, specification of #pragma ADDRESS is invalid.

```
char        port;

void        func(void)
{
            port = 0;           /* Uses a variable before specifying #pragma ADDRESS */
}

#pragma   ADDRESS   port   100H
```

Figure B.73 Cases where the specification of #pragma ADDRESS has no effect

#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 BITADDRESSΔvariable-nameΔbit-position,absolute-address

Description : The absolute address specified in this declaration is expanded as a character string in an assembler file and defined in pseudo instruction .BITEQU. The format for writing the numerical values therefore depends on the assembler, as follows:

(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 a _Bool type variable can be specified to be a variable name. It becomes an error when variables other than _Bool type are specified.
(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) #pragma BITADDRESS is valid for previously declared variables.
(5) #pragma BITADDRESS is invalid if you specify other than a variable.
(6) No error occurs if a #pragma BITADDRESS declaration is duplicated, but the last declared address is valid.
(7) An error occurs if you include an initialization expression.
(8) Normally #pragma BITADDRESS operates on I/O variables, so that even though volatile may not actually be specified, the compiler processes them assuming volatile is specified.
(9) When the -fnot_address_volatile(-fNAV) option is specified, the volatile attribute for the variable defined with #pragma ADDRESS becomes invalid.
(10) If a string that follows "#pragma ADDRESS" consists of white spaces or characters that cannot be a variable name (e.g. 123), or a specified address only consists of white spaces, NC30 outputs a warning message "#pragma ADDRESS format error,ignored" and ignores this statement.
(11) If a string entered as the address contains a character whose 8th bit is 1, NC30 outputs a warning message "Kanji in #pragma ADDRESS" and ignores this statement.

Example :

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

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

Figure B.74 #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-No.∆assembler-function-name (register-name, registername,…)
(2) #pragma INTCALL∆INT-No.∆C-function-name ()

Description : This extended function declares the assembler function called by a software interrupt with the INT number.

Rules :

- Declaring assembler functions
  (1) Before a #pragma INTCALL declaration, be sure to include an assembler function prototype declaration. If there is no prototype declaration, a warning is output and the #pragma INTCALL declaration is ignored.
  (2) Observe the following in the prototype declaration:
    (a) Make sure that the number of parameters in the prototype declaration matches those in the #pragma INTCALL declaration.
    (b) You cannot declare the following types in the parameters in the assembler function:
      - Structure types and union types
      - double types,long double types
      - long long types
    (c) You cannot declare the following functions as the return values of assembler functions:
      - Functions that return structures or unions
  (3) You can use the following registers for parameters when calling:
      - float types, long types (32-bit registers)
        R2R0 and R3R1
      - far pointer types (24-bit registers)
        A0,A1,R2R0, and R3R1
      - 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

      *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
  (1) Before a #pragma INTCALL declaration, be sure to include a prototype declaration. If there is no prototype declaration, a warning is output and the #pragma INTCALL declaration is ignored.
  (2) You cannot specify register names in the parameters of functions that include the #pragma INTCALL declaration.
  (3) Observe the following in the prototype declaration:
    (a) In the prototype declaration, you can only declare functions in which all parameters are passed via registers, as in the function calling rules.
    (d) You cannot declare the following functions as the return values of functions:
      - Functions that return structures or unions
  (4) You can only use decimals for the INT Numbers.

#pragma INTCALL

Declare a function called by the INT instruction

Examples :

```
int         asm_func(unsigned long, unsigned int);       ← Prototype declaration for
#pragma     INTCALL  25  asm_func(R2R0, R1)                the assembler function

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

            i = 0x7FFD;
            l = 0x007F;

            asm_func( l, i );                              ← Calling the assembler function
}
```

Figure B.75 Example of #pragma INTCALL Declaration(asm function) (1)

```
int         c_func(unsigned int, unsigned int);  ← Prototype declaration for the C function
#pragma     INTCALL  25  c_func();               ← You may NOT specify registers.

void        main(void)
{
            int        i, j;

            i = 0x7FFD;
            j = 0x007F;

            c_func( i, j );                      ← Calling the C function
}
```

Figure B.76 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|/V]∆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) By using the above format to declare interrupt processing functions written in C, NC30 generates the code for performing the following interrupt processing at the entry and exit points of 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 of /V 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. When using registers on the back side, be sure that those back registers are not destroyed by an interrupt nest.
- [/E]
  :Multiple interrupts are enabled immediately after entering the interrupt. This improves interrupt response.
- [/V]
  :Generate vector table for fixed vector.

(3) An interrupt vector number can be specified when declaring.

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) You cannot specify both switch /E and switch /B at the same time.
(7) If different interrupt vector numbers are written in the same interrupt handling function, the vector number declared later is effective.
(8) You cannot specify /V and other switches at the same time.

```
#pragma   INTTERUPT   intr(vect=10)
#pragma   INTTERUPT   intr(vect=20)          /* The interrupt vector number 20 is effective. */
```

Figure B.77 Example for writing different interrupt vector numbers

#pragma INTERRUPT

Declare interrupt function

Rules :

(9) A compile warining 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

Example :

```
extern int    int_counter;

#pragma    INTERRUPT    /B    i_func

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

Figure B.78 Example of #pragma INTERRUPT Declaration

Note :

(1) 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 PARAMETER

Declare assembler function that passed arguments via register

Function : Declares an assembler function that passes parameters via registers

Syntax : #pragma PARAMETERΔassembler-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 (24-bit registers) : R2R0, R3R1, A1 and A0
- 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.
- The long long type (64-bit integer type) ,double type and long double type, as well as structure and union types cannot be declared. Furthermore, the following switch can be specified during declaration.

Rules :

(1) Always put the prototype declaration for the assembler function before the #pragma PARAMETER declaration. If you fail to make the prototype declaration, a warning is output and #pragma PARAMETER is ignored.

(2) Follow the following rules in the prototype declaration:

(a) Note also that the number of parameters specified in the prototype declaration must match that in the #pragma PARAMETER declaration.

(b) The following types cannot be declared as parameters for an assembler function in a #pragma PARAMETER declaration:

- structure-type and union-type
- double-type long- long-types long double-type

(c) The assembler functions shown below cannot be declared:

- Functions returning structure or union type

(3) As for the output assembler name of the function specified by #pragma PARAMETER, the _(underscore) is added always previously.

Example :

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

void        main(void)
{
            int         i, j;

            i = 0x7FFD;
            j = 0x007F;

            asm_func( i, j );                            ← Calling the assembler function
}
```

Figure B.79 # 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 Δ[/C]Δ numberΔ function-name()
(2) #pragma SPECIALΔ[/C]Δfunction-name()Δ(number)

Description :
(1) Functions declared using #pragma SPECIAL are mapped to addresses created by adding 0F0000H to the address set in the special page vector tables, and are therefore subject to special page subroutine calls.
(2) You may specify either /C in this declaration:
By specifying switch [/c] it is possible to generate code to need the register to saving it to a stack at entry when calling the function. (only for NC308WA)

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) As a label, "_SPECIAL_calling-number:" is output to the starting address of functions declared using #pragma SPECIAL. Set this label in the special page subroutine table in the startup file.[1]
Note that when the option -fmake_special_table (-fMST) is specified, the above setting is unnecessary.
(4) If different call numbers are written in the function, the call number declared later is effective.

```
#pragma   SPECIAL   func(vect=20)
#pragma   SPECIAL   func(vect=30)          // Call number 30 is effective
```

Figure B.80 Example for writing different call numbers

(5) If functions are defined in one file and function calls are defined in another file, be sure to write this declaration in both files.

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.81 Example of #pragma SPECIAL Declaration

[1] If you are using the supplied startup file, modify the contents of the fvector section. For details of how to modify the startup file, see Chapter 2.2 "Modifying the Startup Program" in the Operation part of the NC30 User's Manual.

B.7.4 The Other Extensions

NC30 includes the following extended function for embedding assembler description inline.

#pragma __ASMMACRO

Assembler macro function

Function : Declares defined a function by assembler macro.

Syntax : #pragma __ASMMACRO . function-name(register name, …)

Rules :

(1) Always put the prototype declaration before the #pragma __ASMMACRO declaration.Assembler macro function be sure to declare "static".

(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.

char type, _Bool type : R0L	float type : R2R0
int type, short type : R0	double type : R3R2R1R0
long type : R2R0	long-long type : R3R1R2R0

(5) If a function call precedes the declaration of #pragma __ASMMACRO, NC30 outputs an error message "#pragma __ASMMACRO must be declared before use".

(6) If #pragma __ASMMACRO is declared for an identifier that is not a function, NC30 outputs a warning message "#pragma __ASMMACRO not function,ignored" and ignores this #pragma statement.

(7) If a functions is declared in a non-prototype form, NC30 outputs a warning message "#pragma __ASMMACRO's function must be prototyped,ignored" and ignores this #pragma statement.

(8) 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.

Example :

```
static long   mul( int, int );                    /* Be sure to declare "static" */

#pragma   __ASMMACRO   mul( R0, R2 )
#pragma   ASM
          _mul         .macro
          mul.w        R2,R0                      ; The return-value is set to R2R0 register
.endm
#pragma   ENDASM

long      l;

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

Figure B.82 Example of #pragma __AMMACRO

#pragma ASM, #pragma ENDASM

Inline assembling

Function : Specifies assembly code in C.

Syntax :

#pragma ASM
assembly statements
#pragma ENDASM

Description : The line(s) between #pragma ASM and #pragma ENDASM are output without modifying anything to the generated assembly source file.
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 assembly-language descriptions, do not write statements which will cause the register contents to be changed. When writing such statements, be sure to use the 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) If the number of characters in one line of assembly-language descriptions exceeds 1024 including the line-feed code, NC30 outputs a warning message "#pragma ASM line too long,then cut" and ignores the 1024th and subsequent characters and line-feed code.

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 area is output directly to an assembly language file.

Figure B.83 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 JSRA

Calls a function with JSR.A

Function : Calls a function using the JSR.A instruction.

Syntax : #pragma JSRA. function-name

Description : Calls all functions declared using #pragma JSRA using the JSR.A instruction. #pragma JSRA can be specified to avoid errors in the case of functions that include code generated using the -fJSRW option and that cause errors during linking.

Rules : This preprocessing directive has no effect when the -fJSRW option not specified.

Example :

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

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

Figure B.84 Example of #pragma JSRA

#pragma JSRW

Calls a function with JSR.W

Function : Calls a function using the JSR.W instruction.

Syntax : #pragma JSRW. function-name

Rules : By default, the JSR.A instruction is used when calling a function that, in the same file, has no body definition. However, the #pragma JSRW-declared function are always called using JSR.W. This directive helps reduce ROM size.

Rules :
(1) You may NOT specify #pragma JSRW for static functions.
(2) When function call with the JSR.W instruction does not reach #pragma JSRW-declared function, an error occurs at link-time. In this case, you may not use #pragma JSRW.

Example :

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

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

Figure B.85 Example of #pragma JSRW

Note : The #pragma JSRW is valid only when directly calling a function. It has no effect when calling indirectly.

#pragma PAGE

Output .PAGE

Function : Declares new-page position in the assembler-generated list file.

Syntax : #pragma PAGE

Description : Putting the line #pragma PAGE in C source code, the .PAGE pseudo-instruction is output at the corresponding line in the compiler-generated assembly source. This instruction causes page ejection asesmbler-output assembly list file.

Rules :
(1) You cannot specify the character string specified in the header of the assembler pseudo-instruction .PAGE.
(2) You cannot write a #pragma PAGE in an auto variable declaration.

Example :

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

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

Figure B.86 Example of #pragma PAGE

B.8 assembler Macro Function

B.8.1 Outline of Assembler Macro Function

NC30 allows part of assembler commands to be written as C-language functions. Because specific assembler commands can be written directly in a C-language program, you can easily tune up the program.

B.8.2 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.

```
#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.87 Description Example of Assembler Macro Function

B.8.3 Commands that Can be Written by Assembler Macro Function

The following shows the assembler commands that can be written using assembler macro functions and their functionality and format as assembler macro functions.

ABS

Function : absolute

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 dadc_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>

static unsigned char divu_b(unsigned char val1, unsigned int val2);
/* 16 bits divided by 8 bits with unsigned */

static 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( signed 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 signed */
```

MOD, MODU

Function: Devide val1 by val2 and get mod.

Syntax : #include <asmmacro.h>

static signed char mod_b(int val1,char val2); /* 16 bits divided by 8 bits with signed */
static signed int mod_w(long val1,int val2); /* 32 bits divided by 16 bits with signed */
static unsigned char modu_b(unsigned int val1,unsigned char val2); /* 16 bits divided by 8 bits with unsigned */
static unsigned int modu_w(unsigned long val1,unsigned int val2);
/* 32 bits divided by 16 bits with unsigned */

NEG

Function : negate

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 */

NOT

Function : not

Syntax : #include <asmmacro.h>
static signed char not_b(signed char val); /* When calculated in 8 bits */
static signed int not_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(singed int init,unsigned int count,signed char *p1,signed char *p2);
/* When calculated in 8 bits */
static long rmpa_w(signed long init, unsigned int count,signed int *p1,signed int *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 *p1,unsigned char *p2,unsigned int count);
/*calculated in 8 bits */
static void smovf_w(unsigned int *p1,unsigned int *p2,unsigned int count);
/*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 24 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 24 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 _far *p1, unsigned char _far *p2, unsigned int count);
/*calculated in 8 bits */
static void smovb_w(unsigned int _far *p1,unsigned int _far *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 _far *p, unsigned int count);
/*calculated in 8 bits */
static void sstr_w(unsigned int val, unsigned int _far *p, unsigned int count);
/*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 */
static unsigned char rot_b( signed char count, unsigned char val );
```

Appendix C   Overview of C Language Specifications

In addition to the standard versions of C available on the market, C language specifications include extended functions for embedded system.

C.1 Performance Specifications

C.1.1 Overview of Standard Specifications

This compiler is a cross C compiler targeting the M16C Series, R8C Family. In terms of language specifications, it is virtually identical to the standard full-set C language, but also has specifications to the hardware in the M16C Series, R8C Family and extended functions for embedded system.

- Extended functions for embedded system(near/far modifiers, and asm function, etc.)
- Floating point library and host machine-dependent functions are contained in the standard library.

C.1.2 Introduction to NC30 Performance

This section provides an overview of NC30 performance.

a. Test Environment

TableC.1 shows the standard PC environment.

TableC.1 Standard PC Environment

Item	Type of PC	OS Version
PC environment	IBM PC/AT or compatible	Windows XP
Type of CPU	Pentium IV	
Memory	128MB min.(Without High-performance Embedded Workshop)	

b. C Source File Coding Specifications

TableC.2 shows the specifications for coding NC30 C source files. Note that estimates are provided for items for which actual measurements could not be achieved.

TableC.2 Specifications for Coding C Source Files

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.

c. NC30 Specifications

TableC.3 to TableC.4 lists the NC30 specifications. Note that estimates are provided for items for which actual measurements could not be achieved.

TableC.3 NC30 Specifications (1)

Item	Specification
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 number of parameters that can be specified in nc30 command line option -ln30	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)

TableC.4 NC30 Specifications (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)

C.2 Standard Language Specifications

The chapter discusses the NC30 language specifications with the standard language specifications.

C.2.1 Syntax

This section describes the syntactical token elements. In NC30, the following are processed as tokens:

- Key words
- Constants
- Operators
- Comment
- Identifiers
- Character literals
- Punctuators

a. Key Words

NC30 interprets the followings as key words.

TableC.5 Key Words List

_asm	_far	_near	asm	auto
_Bool	break	case	char	const
continue	default	do	double	else
enum	extern	far	float	For
goto	if	inline	int	long
near	register	restrict	return	short
signed	sizeof	static	struct	switch
union	unsigned	void	volatile	while
typedef	_ext4mptr	wchar_t	-	-

b. Identifiers

Identifiers consist of the following elements:

- The 1st character is a letter or the underscore (A to Z, a to z, or __)
- The 2nd and subsequent characters are alphanumerics or the underscore (A to Z, a to z, 0 to 9, or __)

Identifiers can consist of up to 200 characters. However, you cannot specify Japanese characters in identifiers.

c. Constants

Constants consists of the followings.

- Integer constants
- Floating point constants
- Character constants

(1) Integer constants

In addition to decimals, you can also specify octal and hexadecimal integer constants. TableC.6 shows the format of each base (decimal, octal, and hexadecimal).

TableC.6 Specifying Integer Constants

Base	Notation	Structure	Example
Decimal	Start with a value other than 0 (zero)	0123456789	15
Octal	Start with 0 (zero)	01234567	017
Hexadecimal	Start with 0X or 0x	0123456789ABCDEF 0123456789abcdef	0XF or 0xf
Binary number	Start with 0b or 0B	01	0b1 or 0B1

Determine the type of the integer constant in the following order according to the value.

- Octal and hexadecimal and Binary number:
  signed int → unsigned int → signed long → unsigned long → signed long long
  → unsigned long long
- Decimal:
  signed int → signed long → signed long long

Adding the suffix U or u, or L or l, or LL or ll, results in the integer constant being processed as follows:

(1) Unsigned constants

Specify unsigned constants by appending the letter U or u after the value. The type is determined from the value in the following order:

unsigned int → unsigned long → unsigned long long

(2) long-type constants

Specify long-type constants by appending the letter L or l. The type is determined from the value in the following order:

- Octal and hexadecimal and Binary number:

signed long → unsigned long → signed long long → unsigned long long

- Decimal :

signed long long → unsigned long long

(3) long long-type constants

Specify long long-type constants by appending the letter LL or ll. The type is determined from the value in the following order:

- Octal and hexadecimal Binary number:

signed long long → unsigned long long

- Decimal :

signed long long

(2) Floating point constants

If nothing is appended to the value, floating point constants are handled as double types. To have them processed as float types, append the letter F or f after the value. If you append L or l, they are treated as long double types.

(3) Character constants

Character constants are normally written in single quote marks, as in 'character'. You can also include the following extended notation (escape sequences and trigraph sequences). Hexadecimal values are indicated by preceding the value with ¥x. Octal values are indicated by preceding the value with ¥.

TableC.7 Extended Notation List

Notation	Escape sequence	Notation	Trigraph sequence
¥'	single quote	¥constant	octal
¥"	quotation mark	¥xconstant	hexadecimal
¥¥	backslash	??(	express "[" character
¥?	question mark	??/	express "¥" character
¥a	bell	??)	express "]" character
¥b	backspace	??'	express "^" character
¥f	form feed	??<	express "{" character
¥n	line feed	??!	express "{" character
¥r	return	??>	express "}" character
¥t	horizontal tab	?? –	express "~" character
¥v	vertical tab	??=	express "#" character

d. Character Literals

Character literals are written in double quote marks, as in "character string". The extended notation shown in TableC.7 for character constants can also be used for character literals.

e. Operators

NC30 can interpret the operators shown in TableC.8.

TableC.8 Operators List

monadic operator	++	logical operator	&&
	−−		¦¦
	−		!
binary operator	+	conditional operator	?:
	−	comma operator	,
	∗	address operator	&
	/	pointer operator	∗
	%	bitwise operator	<<
assignment operators	=		>>
	+=		&
	−=		¦
	∗=		^
	/=		−
	%=		&=
relational operators	>		¦=
	<		^=
	>=		<<=
	<=		>>=
	==	sizeof operator	sizeof
	!=		

f. Punctuators

NC30 interprets the followings as punctuators.

- {
- }
- :
- ;
- ,

g. Comment

Comments are enclosed between / ∗ and ∗/ . They cannot be nested.
Comments are enclosed between "//" and the end of line.

C.2.2 Type

a. Data Type

NC30 supports the following data type.

- character type
- integral type
- structure
- union
- enumerator type
- void
- floating type

b. Qualified Type

NC30 interprets the following as qualified type.

- const
- volatile
- restrict
- near
- far
- _ext4mptr

c. Data Type and Size

TableC.9 shows the size corresponding to data type.

TableC.9 Data Type and Bit Size

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 wchat_t	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	18446744073709551615
float	Yes	32	1.17549435e-38F to 3.40282347e+38F
double long double	Yes	64	2.2250738585072014e-308 to 1.7976931348623157e+308
near pointer	No	16	0 to 0xFFFF
far pointer	No	32	0 to 0xFFFFFFFF

- 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.

C.2.3 Expressions

TableC.10 and TableC.11 show the relationship between types of expressions and their elements.

TableC.10 Types of Expressions and Their Elements (1)

Type of expression	Elements of expression
Primary expression	identifier
	constant
	character literal
	(expression)
	primary expression
Postpositional expression	Postpositional expression [expression]
	Postpositional expression (list of parameters, ...)
	Postpositional expression. identifier
	Postpositional expression -> identifier
	Postpositional expression ++
	Postpositional expression —
	Postpositional expression
Monadic expression	++ monadic expression
	—— monadic expression
	monadic operator cast expression
	sizeof monadic expression
	sizeof (type name)
	Monadic expression
Cast expression	(type name) cast expression
	cast expression
Expression	expression * expression
	expression / expression
	expression % expression
Additional and subtraction expressions	expression + expression
	expression − expression
Bitwise shift expression	expression << expression
	expression >> expression
Relational expressions	expression
	expression < expression
	expression > expression
	expression <= expression
	expression >= expression
Equivalence expression	expression = = expression
	expression != expression
Bitwise AND	expression & expression
Bitwise XOR	expression ^ expression
Bitwise OR	expression | expression
Logical AND	expression && expression
Logical OR	expression | | expression
Conditional expression	expression ? expression: expression

TableC.11　Types of Expressions and Their Elements (2)

Type of expression	Elements of expression
Assign expression	monadic expression += expression
	monadic expression -= expression
	monadic expression *= expression
	monadic expression /= expression
	monadic expression %= expression
	monadic expression <<= expression
	monadic expression >>= expression
	monadic expression &= expression
	monadic expression ¦= expression
	monadic expression ^= expression
	assignment expression
Comma operator	expression, monadic expression

C.2.4 Declaration

There are two types of declaration:

- Variable Declaration
- Function Declaration

a. Variable Declaration

Use the format shown in Figure C.1 to declare variables.

```
storage class specifier. type declarator. declaration specifier. initialization_expression;
```

Figure C.1　Declaration Format of Variable

(1) Storage-class Specifiers

NC30 supports the following storage-class specifiers.

- extern
- static
- typedef
- auto
- register

(2) Type Declarator

NC30 supports the type declarators.

- _Bool
- int
- long
- float
- unsigned
- struct
- enum
- long double
- char
- short
- long long
- double
- signed
- union
- wchar_t

(3) Declaration Specifier

Use the format of declaration specifier shown in Figure C.2 in NC30.

```
Declarator : Pointer opt declarator2
Declarator2 : identifier( declarator )
              declarator2[ constant expression opt ]
              declarator2( list of dummy arguments opt )
* Only the first array can be omitted from constant expressions showing the number of arrays.
* opt indicates optional items.
```

Figure C.2 Format of Declaration Specifier

(4) Initialization expressions

NC30 allows the initial values shown in Figure C.3 in initialization expressions.

```
integral types : constant
integral types array : constant, constant ....
character types : constant
character types array : character literal, constant ....
pointer types : character literal
pointer array : character literal, character literal ....
```

Figure C.3 Initial Values Specifiable in Initialization Expressions

b. Function Declaration

Use the format shown in Figure C.4 to declare functions.

- function declaration (definition)
  storage-class specifier. type declarator. declaration specifier. main program

- function declaration (prototype declaration)
  storage-class specifier. type declarator. declaration specifier;

Figure C.4 Declaration Format of Function

(1) Storage-class Specifier

NC30 supports the following storage-class specifier.

- extern
- static

(2) Type Declarators

NC30 supports the following type declarators.

- _Bool
- int
- long
- float
- unsigned
- struct
- enum
- long double
- char
- short
- long long
- double
- signed
- union
- wchar_t

(3) Declaration Specifier

Use the format of declaration specifier shown in Figure C.5 in NC30

```
Declarator : Pointer opt declarator2
Declarator2 : identifier( list of dummy argument opt )
              ( declarator )
              declarator[ constant expressiono opt]
              declarator( list of dummy argument opt )

* Only the first array can be omitted from constant expressions showing the number of arrays.
* opt indicates optional items.
* The list of dummy arguments is replaced by a list of type declarators in a prototype declaration.
```

Figure C.5 Format of Declaration Specifier

(4) Body of the Program

Use the format of body of the program shown in Figure C.6

```
List of Variable Declaratoropt Compound Statement

*There is no body of the program in a prototype declaration, which ends with a semicolon.
*opt indicates optional items.
```

Figure C.6 Format of Body of the Program

C.2.5 Statement

NC30 supports the following.

- Labelled Statement
- Expression / Null Statement
- Iteration Statement
- Assembly Language Statement
- Compound Statement
- Selection Statement
- Jump Statement

a. Labelled Statement

Use the format of labelled statement shown in Figure C.7

```
Identifier : statement
case constant : statement
default : statement
```

Figure C.7 Format of Labelled Statement

b. Compound Statement

Use the format of compound statement shown in Figure C.8

```
{ list of declarationsoptlist of statementsopt opt }
* opt indicates optional items.
```

Figure C.8 Format of Compound Statement

c. Expression / Null Statement

Use the format of expression and null statement shown in Figure C.9

```
expression:
expression;
null statement:
;
```

Figure C.9 Format of Expression and Null Statement

d. Selection Statement

Use the format of selection statement shown in Figure C.10

```
if( expression )statement
if( expression )statement else statement
switch( expression )statement
```

Figure C.10 Format of Selection Statement

e. Iteration Statement

Use the format of iteration statement shown in Figure C.11

```
while( expression )statement
do statement while ( expression );
for( expression opt;expression opt;expression opt )statement;

* opt indicates optional items.
```

Figure C.11 Format of Iteration Statement

f. Jump statement

Use the format of jump statement shown in Figure C.12

```
goto identifier;
continue;
break;
return expression opt;

*opt indicates optional items.
```

Figure C.12 Format of Jump Statement

g. Assembly Language Statement

Use the format of assembly language shown in Figure C.13

```
asm( "Literals" );
literals : assembly language statement
```

Figure C.13 Format of Assembly Language Statement

C.3 Preprocess Commands

Preprocess commands start with the pound sign (#) and are processed by the cpp30 preprocessor. This chapter provides the specifications of the preprocess commands.

C.3.1 List of Preprocess Commands Available

TableC.12 lists the preprocess commands available in NC30.

TableC.12 List of Preprocess Commands

Command	Function
#assert	Outputs a warning when a constant expression is false.
#define	Defines macros.
#elif	Performs conditional compilation.
#else	Performs conditional compilation.
#endif	Performs conditional compilation.
#error	Outputs messages to the standard output device and terminates processing.
#if	Performs conditional compilation.
#ifdef	Performs conditional compilation.
#ifndef	Performs conditional compilation.
#include	Takes in the specified file.
#line	Specifies file's line numbers.
#pragma	Instructs processing for this compiler extended function.
#undef	Undefines macros.

C.3.2 Preprocess Commands Reference

The NC30 preprocess commands are described in more detail below. They are listed in the order shown in TableC.12.

#assert

Function: Issues a warning if a constant expression results in zero (0).

Format: #assert constant expression

Description: Issues a warning if a constant expression results in zero (0). Compile is continued, however.

```
[Warning(cpp30.82):x.c, line xx]assertion warning
```

#define

Function: Defines macros.

Format:
(1) #define identifier lexical string opt
(2) #define identifier (identifier list opt) lexical string opt

Description:
(1) Defines an identifier as macro.
(2) Defines an identifier as macro. In this format, do not insert any space or tab between the first identifier and the left parenthesis '('.

- The identifier in the following code is replaced by blanks.

```
#define   SYMBOL
```

- When a macro is used to define a function, you can insert a backslash so that the code can span two or more lines.
- The following four identifiers are reserved words for the compiler.

```
__FILE__ .............................. Name of source file
__LINE__ .............................. Current source file line No.
__DATE__ .............................. Date compiled (mm dd yyyy)
__TIME__ .............................. Time compiled (hh:mm:ss)
```

The following are predefined macros in NC30.

```
M16C ( As for the time of "-R8C" option and –R8CE use, __R8C__ is defined instead. )
 NC30
__NEAR__(コンパイルオプション-R8C 使用時のみ)
```

- You can use the token string operator '#' and token concatenated operator '##' with tokens, as shown below.

```
#define debug(s,t) printf("x"#s" = %d x"#t" = %d",x ## s,x ## t)
When parameters are specified for this macro debug (s, t) as debug (1, 2), they are interpreted as
follows:
#define debug(s,t) printf("x1 = %d x2 = %d", x1,x2)
```

- Macro definitions can be nested (to a maximum of 20 levels) as shown below.

```
#define   XYZ1   100
#define   XYZ2   XYZ1
                :
          (abbreviated)
                :
#define   XYZ20   XYZ19
```

#error

Function: Suspends compilation and outputs the message to the standard output device.

Format: #error character string

Description:
- Suspends compilation.
- lexical string is found, this command outputs that character string to the standard output device.

#if - #elif - #else - #endif

Function: Performs conditional compilation. (Examines the expression true or false.)

Format:
```
#if constant expression
:
#elif constant expression
:
#else
:
#endif
```

Description:
- If the value of the constant is true (not 0), the commands #if and #elif process the program that follows.
- #elif is used in a pair with #if, #ifdef, or #ifndef.
- #else is used in a pair with #if. Do not specify any tokens between #else and the line feed. You can, however, insert a comment.
- #endif indicates the end of the range controlled by #if. Always be sure to enter #endif when using command #if.
- Combinations of #if - #elif - #else - #endif can be nested. There is no set limit to the number of levels of nesting (but it depends on the amount of available memory).

#ifdef - #elif - #else - #endif

Function: Performs conditional compilation. (Examines the macro defined or not.)

Format:

```
#ifdef identifier
:
#elif constant expression
:
#else
:
#endif
```

Description:

- If an identifier is defined, #ifdef processes the program that follows. You can also describe the following.

```
#if defined identifier
#if defined (identifier)
```

- #else is used in a pair with #ifdef. Do not specify any tokens between #else and the line feed. You can, however, insert a comment.
- #elif is used in a pair with #if, #ifdef, or #ifndef.
- #endif indicates the end of the range controlled by #ifdef. Always be sure to enter #endif when using command #ifdef.
- Combinations of #ifdef - #else - #endif can be nested. There is no set limit to the number of levels of nesting (but it depends on the amount of available memory).

#ifndef - #elif - #else - #endif

Function: Performs conditional compilation. (Examines the macro defined or not.)

Format:

```
#ifndef identifier
:
#elif constant expression
:
#else
:
#endif
```

Description:

- If an identifier isn't defined, #ifndef processes the program that follows. You can also describe the followings.

```
#if !defined identifier
#if !defined (identifier)
```

- #else is used in a pair with #ifndef. Do not specify any tokens between #else and the line feed. You can, however, insert a comment.
- #elif is used in a pair with #if, #ifdef, or #ifndef.
- #endif indicates the end of the range controlled by #ifndef. Always be sure to enter #endif when using command #ifndef.
- Combinations of #ifndef - #else - #endif can be nested. There is no set limit to the number of levels of nesting (but it depends on the amount of available memory).
- You cannot use the sizeof operator, cast operator, or variables in a constant expression.

#include

Function: Takes in the specified file.

Format:
(1) #include <file name>
(2) #include "file name"
(3) #include identifier

Description:
(1) Takes in <file name> from the directory specified by nc30's command line option -I. Searches <file name> from the directory specified by environment variable "INC30" if it's not found.
(2) Takes in "file name" from the current directory. Searches "file name" from the following directory in sequence if it's not found.
    (1) The directory specified by nc30's startup option -I.
    (2) The directory specified by environment variable "INC30"
(3) If the macro-expanded identifier is <file name> or "file name" this command takes in that file from the directory according to rules of search [1] or [2].

- The maximum number of levels of nesting is 40.
- An include error results if the specified file does not exist.

#line

Function: Changes the line number in the file.

Format: #line integer "file name"

Description:
- Specify the line number in the file and the file name.
- You can change the name of the source file and the line No.

#pragma

Function: Instructs the system to process NC30's extended functions.

Format:

(1) #pragma ROM variable name
(2) #pragma SBDATA variable name
(3) #pragma SECTION predetermined section name. altered section name
(4) #pragma STRUCT tag name of structure unpack
(5) #pragma STRUCT tag name of structure arrange
(6) #pragma EXT4MPTR name of pointer
(7) #pragma ADDRESS variable name absolute address
(8) #pragma BITADDRESS variable name bit position, absolute address
(9) #pragma INTCALL int No.. assembler function name(register name, register name, ..)
(10) #pragma INTCALL int No.. C language function name()
(11) #pragma INTERRUPT [/B|/E] interrupt handling function name
(12) #pragma PARAMETER assembler function name(register name, register name, ..)
(13) #pragma SPECIAL special No.. function name
(14) #pragma ASM
(15) #pragma ENDASM
(16) #pragma JSRA function name
(17) #pragma JARW function name
(18) #pragma PAGE
(19) #pragma __ASMMACRO function name(register name)

#pragma

Description:

(1) Facility to arrange in the rom section
(2) Facility to describe variables using SB relative addressing
(3) Facility to alter the section base name
(4) Facility to control the array of structures
(5) Facility to control the array of structures
(6) Facility to declare pointer for access 4M-byte ROM area
(7) Facility to specify absolute addresses for input/output variables
(8) Facility to specify absolute-with bit position addresses for input/output variables
(9) Facility to declare functions using software interrupts
(10) Facility to declare functions using software interrupts
(11) Facility to write interrupt functions
(12) Facility to declare assembler functions passed via register
(13) Facility to declare special page subroutine call functions
(14) Facility to describe inline assembler
(15) Facility to describe inline assembler
(16) Facility to declare functions calling with JSR.A instruction
(17) Facility to declare functions calling with JSR.W instruction
(18) Facility to output .PAGE
(19) Facility to declare Assembler macro function

- You can only specify the above 24 processing functions with #pragma. If you specify a character string or identifier other than the above after #pragma, it will be ignored.
- By default, no warning is output if you specify an unsupported #pragma function. Warnings are only output if you specify the nc30 command line option -Wunknown_pragma (-WUP).

#undef

Function: Nullifies an identifier that is defined as macro.

Format: #undef identifier

Description:

- Nullifies an identifier that is defined as macro.
- The following four identifiers are compiler reserved words. Because these identifiers must be permanently valid, do not undefine them with #undef.

```
__FILE__ .............................Name of source file
__LINE__ .............................Current source file line No.
__DATE__ ...........................Date compiled (mm dd yyyy)
__TIME__ ............................Time compiled (hh:mm:ss)
```

C.3.3 Predefined Macros

The following macros are predefined in NC30:

- M16C (As for the time of "-R8C" option and –R8CE option use, _ _R8C_ _ is defined instead.)
- NC30
- __NEAR__(only when the compile option -R8C is in use)

C.3.4 Usage of predefined Macros

The predefined macros are used to, for example, use preprocess commands to switch machine-dependent code in non-NC30 C programs.

```
#ifdef  NC30
#pragma  ADDRESS  port0  2H
#pragma  ADDRESS  port1  3H
#else
#pragma  AD  portA = 0x5F
#pragma  AD  portA = 0x60
#endif
```

Figure C.14 Usage Example of Predefined Macros

Appendix D 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 Internal Representation of Data

D.1.1 Integral Type

Table D.1 shows the number of bytes used by integral type data

Table D.1 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
wchar_t	No	16	0 to 65535
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	18446744073709551615
near pointer	No	16	0 to 0xFFFF
far pointer	No	32	0 to 0xFFFFFFFF

- 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.

D.1.2 Floating Type

Table D.2 shows the number of bytes used by floating type data.

Table D.2 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-30 to 1.7976931348623157e+30

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.

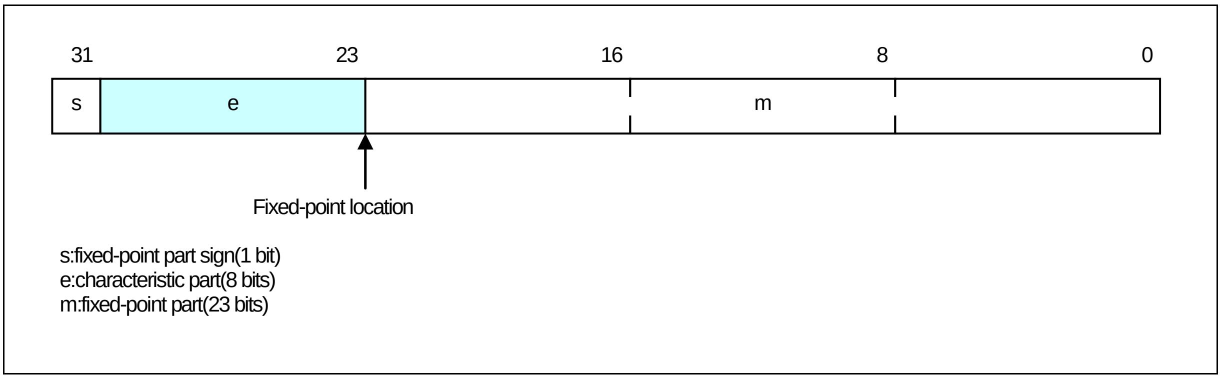


Figure D.1 Single-precision floating point data format

(2) Double-precision floating point data format

Figure D.2 shows the format for binary floating point (double and long double) data.

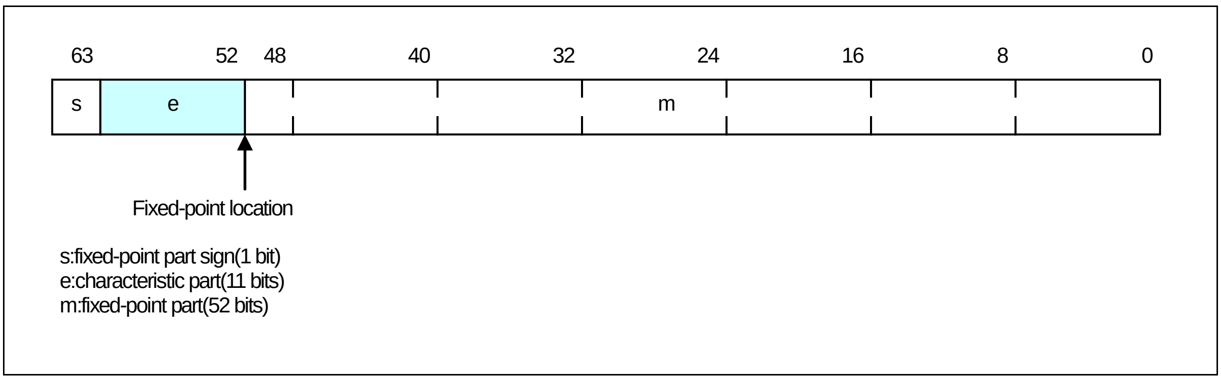


Figure D.2 Double-precision floating point data format

D.1.3 Enumerator Type

Enumerator types have the same internal representation as unsigned int types. Unless otherwise specified, integers 0, 1, 2, are applied in the order in which the members appear.
Note that you can also use the nc30 command line option -fchar_enumerator (-fCE) to force enumerator types to have the same internal representation as unsigned char types.

D.1.4 Pointer Type

Table D.3 shows the number of bytes used by pointer type data.

Table D.3 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.1.5 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.

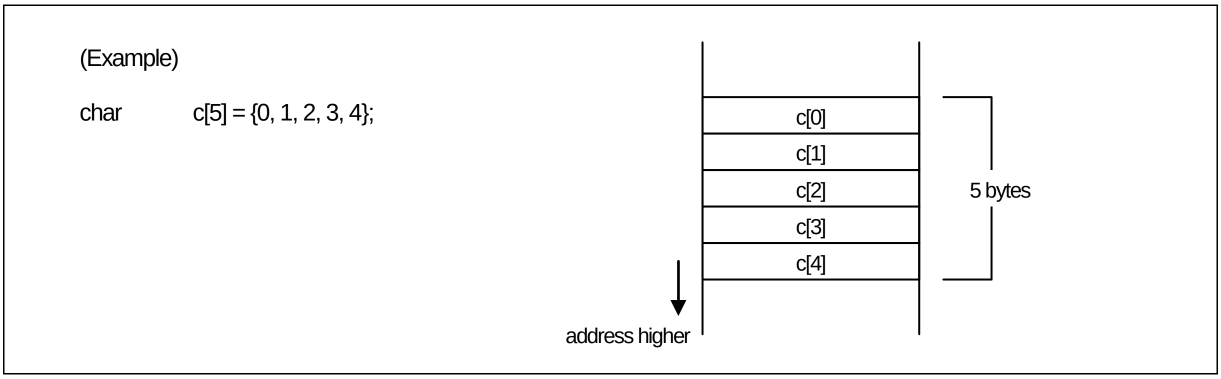


Figure D.3 Example of Placement of Array

D.1.6 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;
```

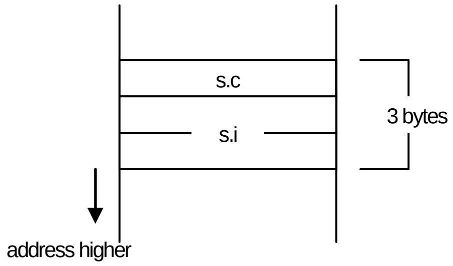


Figure D.4 Example of Placement of Structure (1)

Normally, there is no word alignment with structures. The members of structures aremapped 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   unpac

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

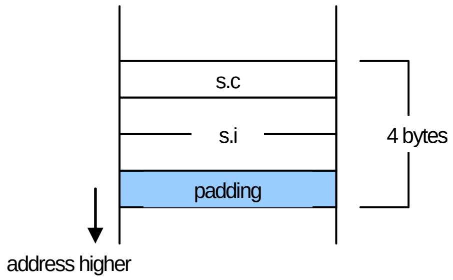


Figure D.5 Example of Placement of Structure (2)

D.1.7 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;
```

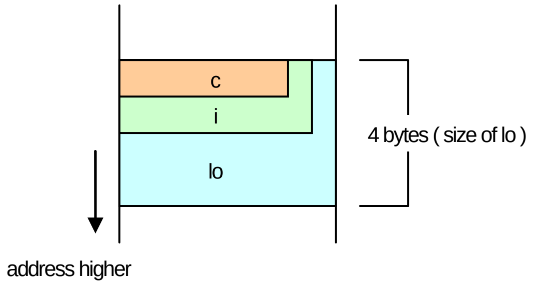


Figure D.6 Example of Placement of Union

D.1.8 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;
```

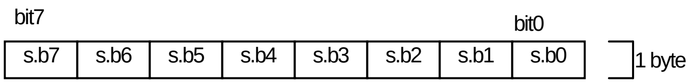


Figure D.7 Example of Placement of Bitfield (1)

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;
```

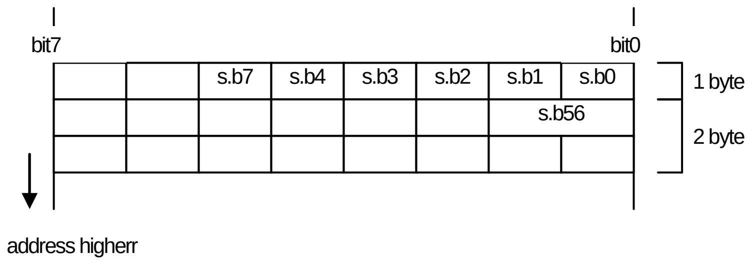


Figure D.8 Example of Placement of Bitfield (2)

- 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 Sign Extension Rules

Under the ANSI and other standard C language specifications, char type data is sign extended to int type data for calculations, etc. This specification prevents the maximum value for char types being exceeded with unexpected results when performing the char type calculation shown in Figure D.9

```
void        func(void)
{
            char        c1, c2, c3;

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

Figure D.9 Example of C Program

To generate code that maximizes code efficiency and maximizes speed, NC30 does not, by default, extend char types to int types. The default can, however, be overridden using the nc30 compile driver command line option -fansi or -fextend_to_int (-fETI) to achieve the same sign extension as in standard C.
If you do not use the -fansi or -fextend_to_int (-fETI) option and your program assigns the result of a calculation to a char type, as in Figure D.9 make sure that the maximum or minimum[1] value for a char type does not result in an overflow in the calculation.

[1] The ranges of values that can be expressed as char types in NC30 are as follows:
* unsigned char type 0. 255,
* signed char type .... -128. 127

D.3 Function Call Rules

D.3.1 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.4 shows rules on calls regarding return values.

Table D.4 Return Value-related Calling Rules

Type of Return Value	Rules
_Boll char	R0L Register
int near pointer	R0 Register
float long far pointer	Least significant 16 bits returned by storing in R0 register. Most significant 16 bits returned by storing in R2 register.
double long double	Values are stored in 16 bits beginning with the high-order bits sequentially in order of registers R3, R2, R1, and R0 as they are returned.
long long	Values are stored in 16 bits beginning with the high-order bits sequentially in order of registers R3, R1, R2, and R0 as they are returned.
Structure Type Union Type	Immediately before the function call, save the far address for the area for storing the return value to the stack. Before execution returns from the called function, that function writes the return value to the area indicated by the far address saved to the stack.

D.3.2 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.5 and Table D.6 to pass arguments.

- Function is prototype declared[1] and the type of argument is known when calling the function.
- Variable argument "..." is not used in prototype declaration.
- For the type of the argument of a function, the Argument and Type of Argument in Table D.5 and Table D.6 are matched.

Table D.5 Rules on Argument Transfer via Register (NC308)

Argument	First Argument	Registers Used
First argument	char type, _Bool type	R0L register
	int type near pointer type	R0 register

[1] NC30 uses a via-register transfer only when entering prototype declaration (i.e., when writing a new format). Consequently, all arguments are passed via stack when description of K&R format is entered (description of old format).
Note also that if a description format where prototype declaration is entered for the function (new format) and a description of the K&R format (old format) coexist in given statement, the system may fail to pass arguments to the function correctly, for reasons of language specifications of the C language.
Therefore, we recommends using a prototype: declaring description format as the standard format to write the C language source files for NC30.

Table D.6 Rules on Argument Transfer via Register (NC30)

Argument	First Argument	Registers Used
First argument	char type, _Bool type	R1L register
	int type near pointer type	R1 register
Second argument	int type near pointer type	R2 register

(2) Passing arguments via stack

All arguments that do not satisfy the register transfer requirements are passed via stack. The Table D.7 and Table D.8 summarize the methods used to pass arguments.

Table D.7 Rules on Passing Arguments to Function (NC308)

Type of Argument	First Argument	Second Argument	Third and Following Arguments
char type _Bool type	R0L register	Stack	Stack
int type near pointer type	R0 register	Stack	Stack
Other types	Stack	Stack	Stack

Table D.8 Rules on Passing Arguments to Function (NC30)

Type of Argument	First Argument	Second Argument	Third and Following Arguments
char type _Bool type	R1L register	Stack	Stack
int type near pointer type	R1 register	R2 registe	Stack
Other types	Stack	Stack	Stack

D.3.3 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 start label of a function in an assembler source file consists of the function name in the C language source file prefixed by _ (underbar) or $ (dollar).
The table below lists the character strings that are added to a function name and the conditions under which they are added.

Table D.9 Conditions Under Which Character Strings Are Added to Function

Added character string	Condition
$ (dollar)	Functions where any one of arguments is passed via register
_ (underbar)	Functions that do not belong to the above[1]

Shown in Figure D.10 is a sample program where a function has register arguments and where a function has its arguments passed via only a stack.

[1] However, function names are not output for the functions that are specified by #pragma INTCALL.

```
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] This is the prototype declaration of function func_proto.
[2] This is the body of function func_proto. (Prototype declaration is entered, so this is a new format.)
[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.10 Sample Program for Calling a Function (sample.c)

The compile result of the above sample program is shown in the next page. Figure D.11 shows the compile result of program part[2]that defines function func_proto. Figure D.12 shows the compile result of program part[3]that defines function func_no_proto.Figure D.13 shows the compile result of program part[4]that calls function func_proto and function func_no_proto.

```
;## #        FUNCTION func_proto
;## #        FRAME      AUTO       (          j)   size    2,      offset -4
;## #        FRAME      AUTO       (          i)   size    2,      offset -2
;## #   FRAME      ARG (          k)   size     2,      offset 5                    ← [7]
;## #   REGISTER ARG       (          i)    size     2,     REGISTER R1            ← [8]
;## #   REGISTER ARG       (          j)    size     2,     REGISTER R2            ← [9]
;## #        ARG Size(2)              Auto Size(2)            Context Size(5)


             .SECTION  program,CODE,ALIGN
             ._file       'sample.c'
             .align
             ._line       4
;## # C_SRC :            {
             .glb         $func_proto
$func_proto:                                                                     ← [10]
             enter        #04H
             mov.w        R1,-2[FB]   ;   i   i
             mov.w        R2,-4[FB]   ;   j   j
             ._line       5
;## # C_SRC :                         return i + j + k;
             mov.w        -2[FB],R0   ;   i
             add.w        -4[FB],R0   ;   j
             add.w        5[FB],R0    ;   k
             exitd
E1:
```

This passes the first argument j via register.

[7] This passes the third argument k via stack.
[8] This passes the first argument j via register.
[9] This passes the secondargument i via register.
[10] This is the start address of function func_proto.

Figure D.11 Compile Result of Sample Program (sample.c) (1)

In the compile result (1) of the sample program (sample.c) listed in Figure D.10, the first and second arguments are passed via a register since function func_proto is prototype declared. The third argument is passed via a stack since it is not subject to via-register transfer.
Furthermore, since the arguments of the function are passed via register, the symbol name of the function's start address is derived from "func_proto" described in the C language source file by prefixing it with $ (dollar), hence, "$func_proto."

```
;## #        FUNCTION func_no_proto
;## #   FRAME    ARG (        i)   size    2,     offset 5        [11]
;## #   FRAME    ARG (        j)   size    2,     offset 7
;## #   FRAME    ARG (        k)   size    2,     offset 9
;## #        ARG Size(6)          Auto Size(0)          Context Size(5)

             .align
             ._line      12
;## # C_SRC :            {
             .glb        _func_no_proto                    ←[12]
_func_no_proto:
             enter       #00H
             ._line      13
;## # C_SRC :                     return i + j + k;
             mov.w       5[FB],R0    ;   i
             add.w       7[FB],R0    ;   j
             add.w       9[FB],R0    ;   k
             exitd
E2:


[11] This passes all arguments via a stack.
[12] This is the start address of function func_no_proto.
```

Figure D.12 Compile Result of Sample Program (sample.c) (2)

In the compile result (2) of the sample program (sample.c) listed in Figure D.10, all arguments are passed via a stack since function func_no_proto is written in K&R format.
Furthermore, since the arguments of the function are not passed via register, the symbol name of the function's start address is derived from "func_no_proto" described in the C language source file by prefixing it with _ (underbar), hence, "_func_no_proto."

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

            .align
            ._line    17
;###C_SRC :           {
            .glb      _main
_main:
            enter     #02H
            ._line    20
;###C_SRC :                     sum = func_proto(1,2,3);
            push.w    #0003H                                          [13]
            mov.w     #0002H,R2
            mov.w     #0001H,R1
            jsr       $func_proto
            add.l     #02H,SP
            mov.w     R0,-2[FB]  ;  sum
            ._line    21
;###C_SRC :                     sum = func_no_proto(1,2,3);
            push.w    #0003H                                          [14]
            push.w    #0002H
            push.w    #0001H
            jsr       _func_no_proto
            add.l     #06H,SP
            mov.w     R0,-2[FB]  ;  sum
            ._line    22
;###C_SRC :           }
            exitd
E3:
            .END
```

Figure D.13 Compile Result of Sample Program (sample.c) (3)

Figure D.13 ,part[13]calls func_proto and part[14]calls func_no_proto.

D.3.4 Interface between Functions

Figure D.16 to D.18 show the stack frame structuring and release processing for the program shown in Figure D.14. Figure D.15 shows the assembly language program that is produced when the program shown in Figure D.14 is compiled.

```
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.14 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'
          .align
          ._line    4
;###C_SRC :          {
          .glb      _main
_main:                                                            ← [1]
          enter     #06H                                          ← [2]
          ._line    5
;###C_SRC :                    int         i = 0x1234;
          mov.w     #1234H,-2[FB]          ;  i
          ._line    6
;###C_SRC :                    int         j = 0x5678;
          mov.w     #5678H,-4[FB]          ;  j
          ._line    7
;###C_SRC :                    int         k = 0x9abc;
          mov.w     #9abcH,-6[FB]          ;  k
          ._line    9
;###C_SRC :                    k = func( i, j ,k);
          push.w    -6[FB]     ;  k                               ← [3]
          mov.w     -4[FB],R2  ;  j                               ← [4]
          mov.w     -2[FB],R1  ;  i                               ← [5]
          jsr       $func                                         ← [6]
          add.l     #02H,SP                                       ← [10]
          mov.w     R0,-2[FB]  ;  k                               ← [11]
          ._line    10
;###C_SRC :          }
          exitd
E1:
```

Figure D.15 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)

            .align
            ._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:
            .END
```

Figure D.16 Assembly language sample program (2)

Figure D.16 to D.18 below show stack and register transitions in each processing in Figure D.15. Processing in[1]. [2](entry processing of function main) is shown in Figure D.16. Processing[3]. [4]. [5]. [6]. [7](processing to call function func and construct stack frames used in function func) is shown in Figure D.17.
Processing[8]. [9]. [10]. [11](processing to return from function func to function main) is shown in Figure D.18.

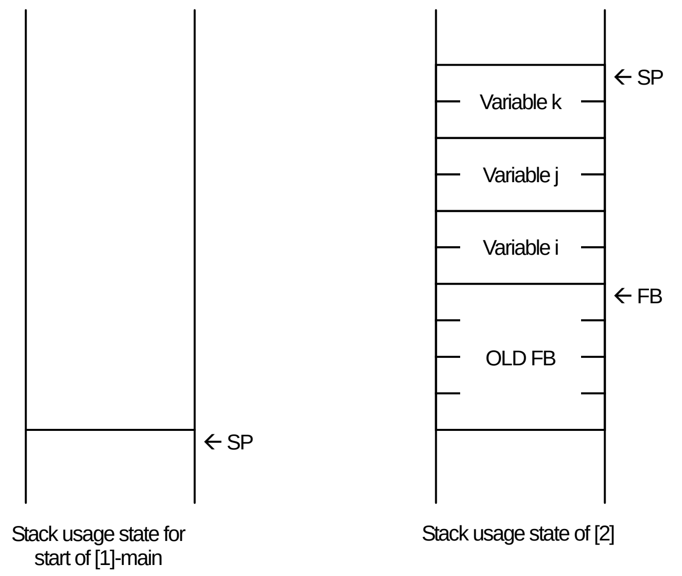


Figure D.17 Entry processing of function main

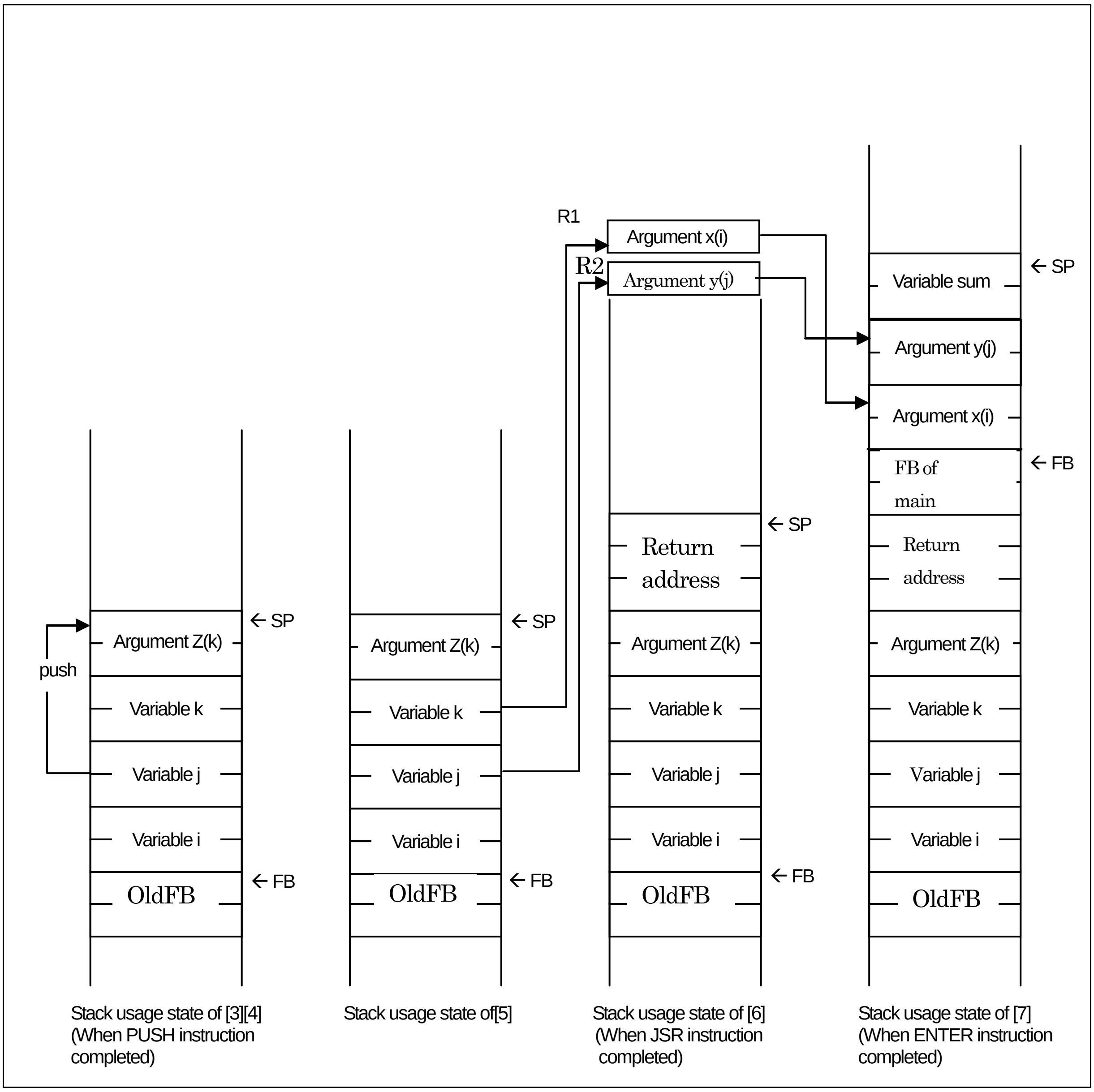


Figure D.18 Calling Function func and Entry Processing

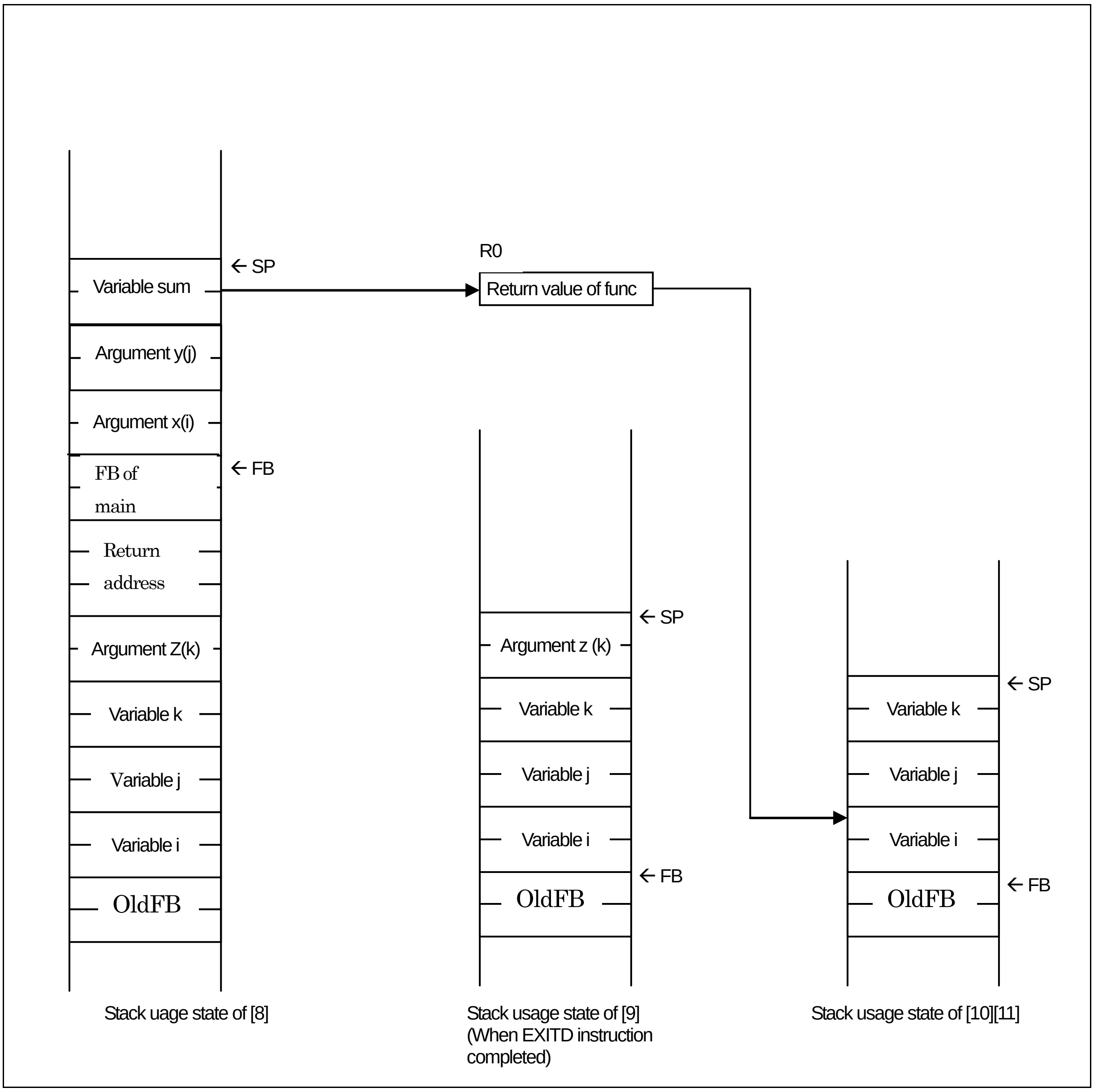


Figure D.19 Exit Processing of Function func

D.4 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.20, 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.20 Example of C Program

In this example, the effective ranges of three auto variables i, j, and k do not overlap, so that a two-byte area (offset 1 from FB) is shared . Figure D.21 shows an assembly language source file generated by compiling the program 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.21 Example of Assembly Language Source Program

D.5 Rules of Escaping of the Register

The rules of Escaping of the register when call C function as follows:

(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

Appendix E Standard Library

E.1 Standard Header Files

When using the NC30 standard library, you must include the header file that defines that function.
This appendix details the functions and specifications of the standard NC30 header files.

E.1.1 Contents of Standard Header Files

NC30 includes the 15 standard header files shown 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 the prototype of input/output functions.
stdlib.h	Declares the prototypes of memory management and terminate functions.
string.h	Declares the prototypes of character string and memory handling 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 assert function.

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	Checks whether the character is a lower-case letter.
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)

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: Defines/declares macros and functions that manipulate program localization.The following lists locale functions.

Function	Contents
localeconv	Initializes struct lconv.
setlocale	Sets and searches the locale information of a program.

math.h

Function: Declares prototype of mathematical function.The following lists mathematical functions.

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.
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.

mathf.h

Function: Includes math.h and defines a macro that validates single-precision mathematical functions.

setjmp.h

Function: Defines the structures used in branch functions.

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/declares the functions which have a variable number of real arguments.

stddef.h

Function: Defines the macro names which are shared among standard include files.

stdio.h

Function: Defines the FILE structure,stream name, and declares I/O function prototypes. Prototype declarations are made for the following functions.

Type	Function	Function
Initialize	init	Initializes R8C family input/outputs.
	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 the prototypes of memory management and terminate functions.

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 the prototypes of string handling functions and memory handling functions.

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

NC30 provides a standard function library covering most functions. These functions are classified into the following categories.

(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	Converts a character string (using locale information).	○

* Several standard functions use global variables that are specific to that function. If, while that function is called and is being executed, an interrupt occurs and that same function is called by the interrupt processing program, the global variables used by the function when first called may be overwritten.

This does not occur to global variables of functions with reentrancy (indicated by a O in the table). However, if the function does not have reentrancy (indicated by a X in the table), care must be taken if the function is also used by an interrupt processing program.

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	Checks whether the character is a lower-case letter.	○
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	_init	Initializes M16C Series's input/outputs.	×
	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.	○
acosf	Calculates single-precision arc cosine	○
asinf	Calculates single-precision arc sine	○
atanf	Calculates single-precision arc tangent	○
atan2f	Calculates single-precision arc tangent	○
ceilf	Calculates single-precision an integer carry value	○
cosf	Calculates single-precision consine	○
coshf	Calculates single-precision hyperbolic cosine	○
expf	Calculates single-precision exponential function	○
fabsf	Calculates single-precision absolute value	○
floorf	Calculates single-precision borrow value	○
fmodf	Calculates single-precision remainder	○
frexpf	Divides floating-point number into mantissa and exponent parts	○
logf	Calculates single-precision natural logarithm	○
log10f	Calculates single-precision common logarithm	○
modff	Calculates the division of a real number into the mantissa and exponent parts.	○
powf	Calculates single-precision the power of a number.	○
sinf	Calculates single-precision sine	○
sinhf	Calculates single-precision hyperbolic sine.	○

sqrtf	Calculates single-precision the square root of a numeric value.	○
tanf	Calculates single-precision tangent.	○
tanhf	Calculates single-precision 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:	Returns the value derived by converting a character string into a double-precision floating-point number.	

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:	No value is returned.	
Function:	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:	Performs binary search in an array.	
Format:	#include <stdlib.h>	
	void _far *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.
- Interprets code 0x1A as the end code and ignores any subsequent data.
- 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

init

Input/Output Functions

Function: Initializes the stream.

Format: #include <stdio.h>

void init(void);

Method: function

Argument: No argument used.

ReturnValue: No value is returned.

Description:
- Initializes the stream. Also calls speed and init_prn in the function to make the initial settings of the UART and Centronics output device.
- init is normally used by calling it from the startup program.

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; Float-point number
int exp; 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 ....

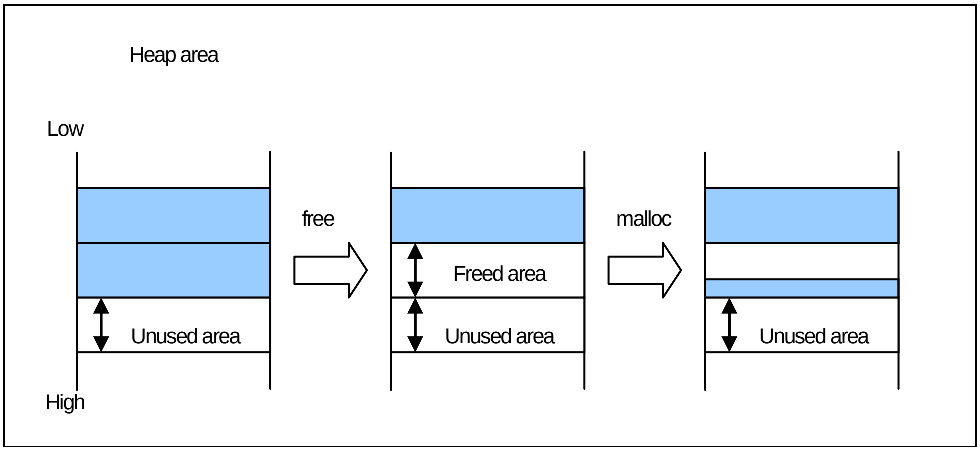
ReturnValue: Returns NULL if a memory area of the specified size could not be allocated.

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.

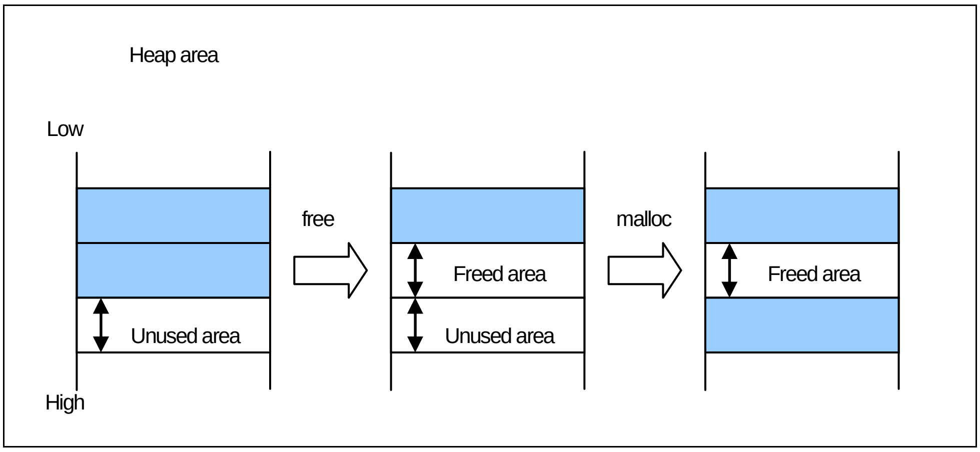


malloc

Memory Management Functions

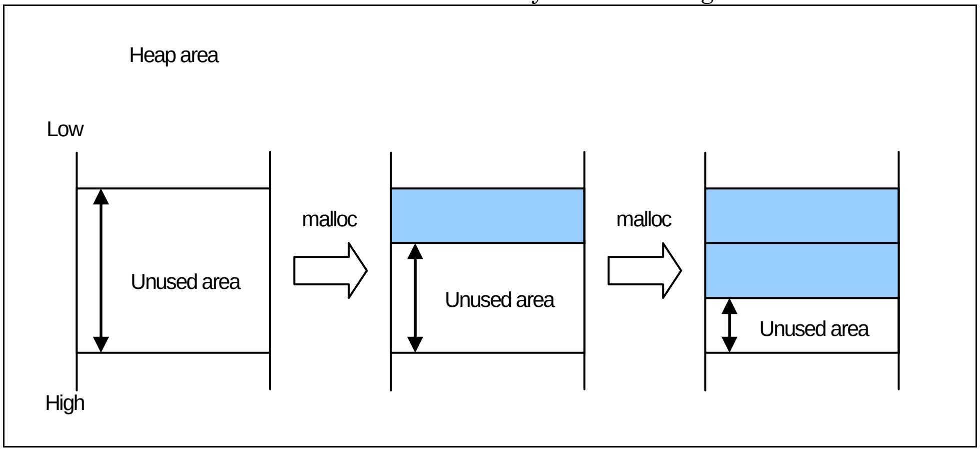
Rule:

- 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".
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM the system may selects another functions with good code efficiency 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
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM the system may selects another functions with good code efficiency by optimization.

memcpy

Memory Handling Functions

Function: Copies n bytes of memory

Format: #include <string.h>

void _far * memcpy(s1, s2, n);

Method: macro(default) or 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:

- Usually, the program code described by macro is used for this function. In using the function in a library, please describe it as #undef memcpy after description of #include <string.h>.
- Copies "n" bytes from memory "S2" to memory "S1".
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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;	Pointer to the first memory area to be compared
char _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 memory areas (with alphabets handled as upper-case letters).
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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 *s1;	Pointer to be moved to
const void *s2;	Pointer to be moved from
size_t n;	Number of bytes to be moved

ReturnValue: Returns a pointer to the destination of movement.

Description: When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency by optimization.

memset

Memory Handling Functions

Function: Set a memory area.

Format: #include <string.h>

void _far * memset(s, c, n);

Method: macro or 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:

- Usually, the program code described by macro is used for this function. In using the function in a library, please describe it as #undef memset after description of #include <string.h>.
- Sets "n" bytes of data "c" in memory "s".
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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; multiplicand
double y; 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 (l, L, or h)] 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.[1]
- 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.

[1] In the standard library included with your product, the character string pointer is a far pointer. (All printf functions handle %s with a far pointer.) Note that scanf functions use a near pointer by default.

printf

Input/Output Functions

Description:

- 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.
- 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.

printf

Input/Output Functions

Description:

- 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.
- 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 you specify an asterisk (*) for the precision, the integer of the parameter specifies the precision.
  (5) If the parameter is a negative value, specification of the precision is invalid.
- 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) I,ll, L or h

- I: d, i, o, u, x, X, and n conversion is performed on long int and unsigned long int parameters.
- h: d, i, o, u, x, and X conversion is performed on short int and unsigned short int parameters.
- Il: d, i, o, u, x, X, or n conversion is performed on long long or unsigned long long parameters. If printf is specified for any other types of conversion, NC30 ignores this specification.
- If I or h are specified in other than d, i, o, u, x, X, or n conversion, they are ignored.
- L: e, E, f, g, and G conversion is performed on double parameters.[1]

(5) When the –R8C option is in use
e, E, f, g, and G conversions are not possible.

[1]In the standard C specifications,variables e,E,f, and g conversions are performed in the case of L on long double parameters .In NC30 ,long double types are processed as double types.Threfore, if you specify L, the parameters are processed as double types.

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: macro

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] [modifier (I, L, or h)]conversion specification character
> 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.

scanf

Input/Output Functions

Description:

(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.
- 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.

scanf

Input/Output Functions

Description:

(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) I, ll,L or h

- I: The results of d, i, o, u, and x conversion are stored as long int and unsigned long int. The results of e, E, f, g, and G conversion are stored as double.
- h: The results of d, i, o, u, and x conversion are stored as short int and unsigned short int.
- Il: d, i, o, u, x, X, or n conversion is performed on long long or unsigned long long parameters. If printf is specified for any other types of conversion, NC30 ignores this specification.
- If I or h are specified in other than d, i, o, u, or x conversion, they are ignored.
- L: The results of e, E, f, g, and G conversion are 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 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.
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency by optimization.

[1] There must be adequate space to accommodate s1 plus s2.

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'.
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency by optimization.

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
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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: When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency by optimization.

strcpy

String Handling Functions

Function: Copies a character string.

Format: #include <string.h>

char _far * strcpy(s1, s2);

Method: macro or 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:
- Usually, the program code described by macro is used for this function. In using the function in a library, please describe it as #undef strcpy after description of #include <string.h>.
- Copies character string "s2" (ending with NULL) to area "s1"
- After copying, the character string ends with NULL.
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects functions with good code efficiency 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.
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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.
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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".
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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.
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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'.
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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'.
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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'.
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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".
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency by optimization.

strtod

Character String Value Convert Functions

Function: Converts a character string into a double-type integer.

Format: #include <string.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: When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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".
- When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency by optimization.

strtol

Character String Value Convert Function

Function: Converts a character string into a long-type integer.

Format: #include <string.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: When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency by optimization.

strtoul

Character String Value Convert Function

Function: Converts a character string into an unsigned long-type integer.

Format: #include <string.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: When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another
functions with good code efficiency 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: When you specify options -O[3 to 5], -OR, -OS, -OR_MAX,-ORM,-OS_MAX, or –OSM, the system may selects another functions with good code efficiency 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: #include <ctype.h>

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
va_list ap;	Pointer of argument 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 _far 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 you specify any of optimization options -O[3 to 5], -OS, -OR, -OR_MAX, -ORM, -OS_MAX, or -ORM, the system performs optimization for the standard functions. This optimization can be suppressed by specifying -Ono_stdlib. Such suppression of optimization is necessary when you use a user function that bear the same name as one of the standard library functions.

(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:far pointer Second argument:string constant	strcpy(str, "sample");
memcpy	First argument:far 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 R8C Family and M16C 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.

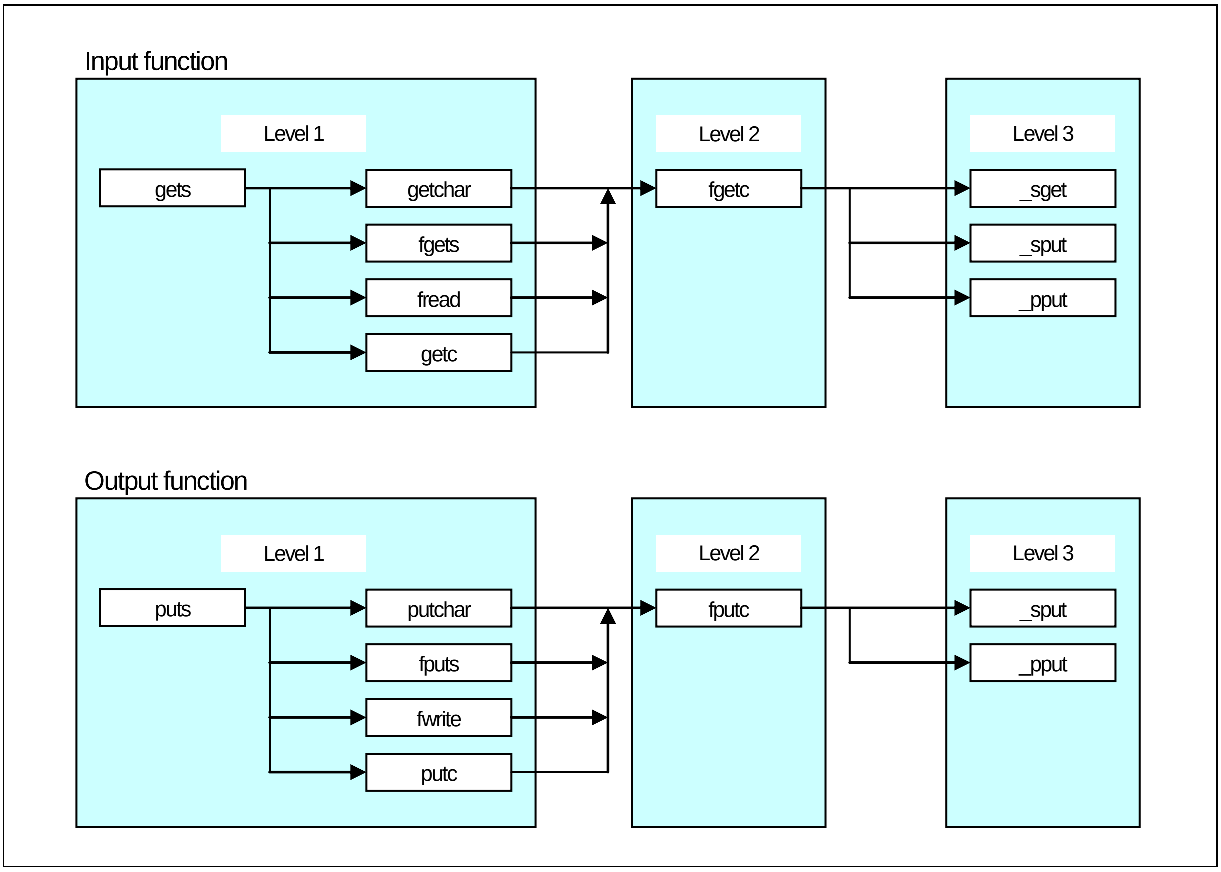


Figure E.1 Calling Relationship of I/O Functions

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.

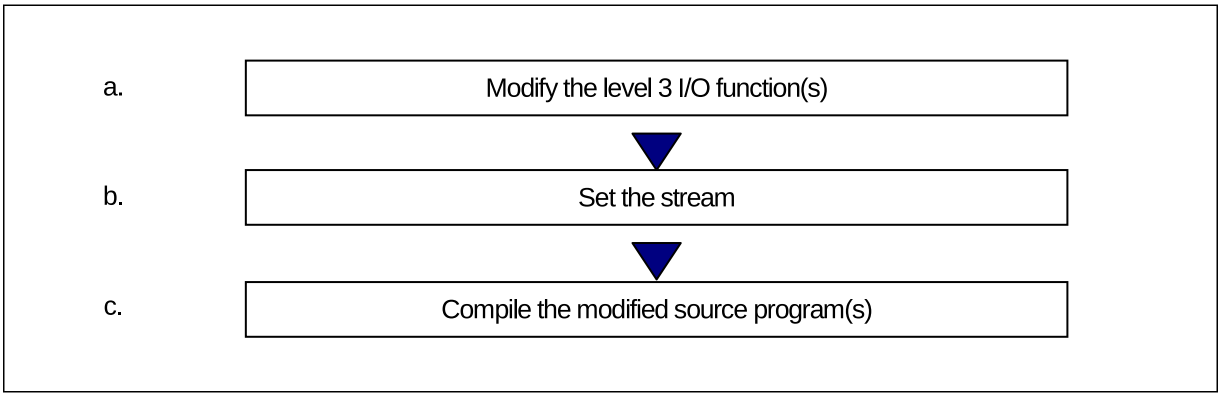


Figure E.2 Example Sequence of Modifying I/O Functions

a. Modifying Level 3 I/O Function

The level 3 I/O functions perform 1-byte I/O via the M16C series R8C Family I/O ports. The level 3 I/O functions include _sget and _sput, which perform I/O via the serial communications circuits (UART), and _pput, which performs I/O via the Centronics communications 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

*The initial serial communications settings are made in the init function (init.c).

The level 3 I/O functions are written in the C library source file device.c. Table E.14 lists the specifications of these functions.

Table E.14 Specifications of Level 3 Functions

Input functions	Parameters	Return value (int type)
_sget _sput _pput	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 communication is set to UART1 in the M16C series's and R8C Family two UARTs. device.c is written so that the UART0 can be selected using the conditional compile commands, as follows:

- To use UART0............................ #define __UART0__

Specify these commands at the beginning of device.c, or specify following option, when compiling.

- To use UART0............................. –D__UART0__

To use both UARTs, modify the file as follows:

(1) Delete the conditional compiling commands from the beginning of the device.c file.

(2) Change the UART0 special register name defined in #pragma ADDRESS to a variable other than UART1.

(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, modify the init function (init.c), which makes the initial I/O function settings, then change the stream settings (see below).

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.

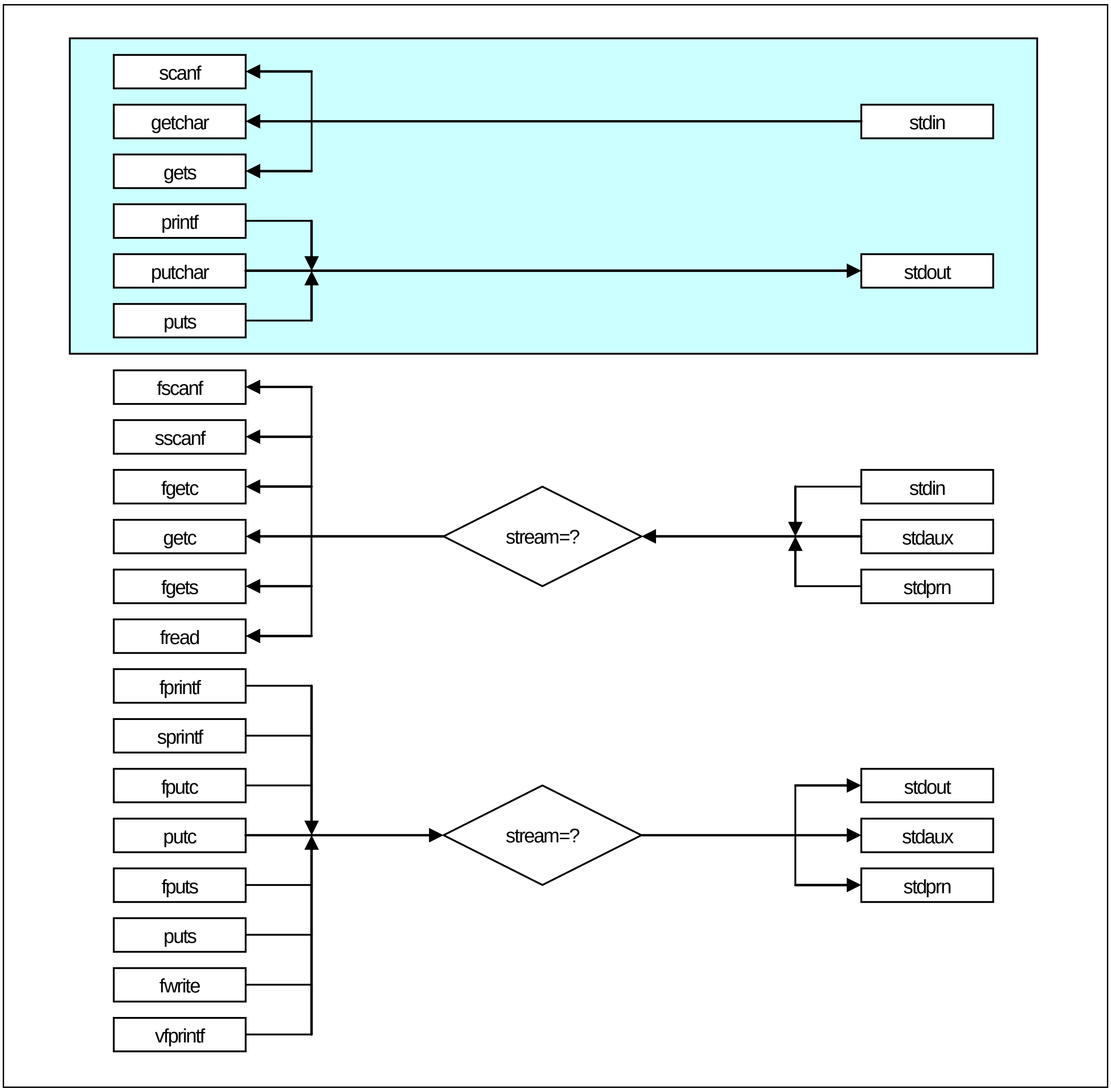


Figure E.3 fRelationship of Functions and Streams

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)()

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)()

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.

Set values for all elements other than char_buff in the stream initialization block. The standard library file supplied in the NC30 package initializes the stream in function init, which is called from the ncrt0.a30 startup program.

Figure E.5 shows the source program for the init function.

```
#include <stdio.h>

FILE _iob[4];

void _init( void );

void _init( void )
{
        stdin->_cnt = stdout->_cnt = stdaux->_cnt = stdprn->_cnt = 0;
        stdin->_flag = _IOREAD;
        stdout->_flag = _IOWRT;
        stdaux->_flag = _IORW;
        stdprn->_flag = _IOWRT;

        stdin->_mod = _TEXT;
        stdout->_mod = _TEXT;
        stdaux->_mod = _BIN;
        stdprn->_mod = _TEXT;

        stdin->_func_in = _sget;
        stdout->_func_in = NULL;
        stdaux->_func_in = _sget;
        stdprn->_func_in = NULL;

        stdin->_func_out = _sput;
        stdout->_func_out = _sput;
        stdaux->_func_out = _sput;
        stdprn->_func_out = _pput;

#ifdef UART0
        speed(_96, _B8, _PN, _S2);
#else /* UART1 : default */
        speed(_96, _B8, _PN, _S2);
#endif
        init_prn();
}
```

Figure E.5 Source file of init function (init.c)

In systems using the two M16C series, R8C Family UARTs, modify the _init function as shown below. In the previous subsection, we set the UART0 functions in the device.c source file temporarily as _sget0, _sput0, and speed0.

(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)
    init_prn();
}

* [2] to [4] correspond to the items in the description of setting, above.
```

Figure E.6 Modifying the init Function

c. Incorporating the Modified Source Program

There are two methods of incorporating the modified source program in the target system:

(1) Specify the object files of the modified function source files when linking.

(2) Use the makefile (under MS-Windows, makefile.dos) supplied in the NC30 package to update the library file.

In method [1], the functions specified when linking become valid and functions with the same names in the library file are excluded.
Figure E.7 shows method(1). Figure E.8 shows method(2).

```
% nc30 -c -g -osample ncrt0.a30 device.r30 init.r30 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

```
% make <RET>
```

Figure E.8 Method of Updating Library Using Modified Source Programs

Appendix F Error Messages

This appendix describes the error messages and warning messages output by this compiler, and their countermeasures.

F.1 Message Format

If, during processing, this compiler detects an error, it displays an error message on the screen and stops the compiling process.
The following shows the format of error messages and warning messages.

```
nc30:[error-message]
```

Figure F.1 Format of Error Messages from the Compile Driver

```
[Error(cpp30.error-No.): filename, line-No.] error-message
[Error(ccom): filename, line-No.] error-message
[Fatal(ccom): filename, line-No.] error-message          ← *1
```

Figure F.2 Format of Command Error Messages

```
[Warning(cpp30. warning-No.): filename, line-No.] warning-message
[Warning(ccom): filename, line-No.] warning-message
```

Figure F.3 Format of Command Warning Messages

*1. Fatal error message
This error message is not normally output. Please contact nearest Renesas office. with details of the message if displayed.

F.2 nc30 Error Messages

Table F.1 and Table F.2 list the nc30 compile driver error messages and their countermeasures.

Table F.1 nc30 Error Messages (1)

Error message	Description and countermeasure
Arg list too long	• The command line for starting the respective processingsystem is longer than the character string defined bythe system. ⇒ Specify a NC30 option to ensure that the number ofcharacters defined by the system is not exceeded. Usethe -v option to check the command line used for eachprocessing block.
Cannot analyze error	• This error message is not normally displayed. (It is aninternal error.) ⇒ Contact Renesas Solutions Corp.
command-file line characters exceed 2048.	• There are more than 2048 characters on one or more lines in the command file. ⇒ Reduce the number of characters per line in the commandfile to 2048 max.
Core dump(command_name)	• The processing system (indicated in parentheses)caused a core dump. ⇒ The processing system is not running correctly. Checkthe environment variables and the directory containingthe processing system. If the processing system stilldoes not run correctly, Please contact Renesas SolutionsCorp.
Exec format error	• Corrupted processing system executable file. ⇒ Reinstall the processing system.
Ignore option '-?'	• You specified an illegal option (-?). ⇒ Specify the correct option.
illegal option	• You specified options greater than 100 characters for -as30 or -ln30. ⇒ Reduce the options to 99 characters or less.
Invalid argument	• It is an internal error. (This error message is not normally displayed.) ⇒ Contact Renesas Solutions Corp.
Invalid option '-?'	• The required parameter was not specified in option "-?". ⇒ "-?"Specify the required parameter after "-?". • You specified a space between the -? option and its parameter. ⇒ Delete the space between the -? option and its parameter.
Invalid option '-o'	• No output filename was specified after the -o option. ⇒ Specify the name of the output file. Do not specify the filename extension.
Invalid suffix '.xxx'	• You specified a filename extension not recognized by NC30 (other than .c, .i, .a30, .r30, .x30). ⇒ Specify the filename with the correct extension.

Table F.2 nc30 Error Messages (2)

Error message	Description and countermeasure
No such file or directory	• The processing system will not run. ⇒ Check that the directory of the processing system is correctly set in the environment variable.
Not enough core	• Insufficient swap area ⇒ Increase the swap area.
Permission denied	• The processing system will not run. ⇒ Check access permission to the processing systems. Or, if access permission is OK, check that the directory of the processing system is correctly set in the environment variable.
can't open command file	• Can not open the command file specified by '@'. ⇒ Specify the correct input file.
too many options	• This error message is not normally displayed. (It is an internal error.) ⇒ Compile options cannot be specified exceeding 99 characters.
Result too large	• It is an internal error. (This error message is not normally displayed.) ⇒ Contact Renesas Solutions Corp.
Too many open files	• It is an internal error. (This error message is not normally displayed.) ⇒ Contact Renesas Solutions Corp.

F.3 cpp30 Error Messages

Table F.3 to Table F.5 list the error messages output by the cpp30 preprocessor and their countermeasures.

Table F.3 cpp30 Error Messages (1)

No.	Error message	Description and countermeasure
1	illegal command option	• Input filename specified twice. ⇒ Specify the input filename once only.
		• The same name was specified for both input and output files. ⇒ Specify different names for input and output files.
		• Output filename specified twice. ⇒ Specify the output filename once only.
		• The command line ends with the -o option. ⇒ Specify the name of the output file after the –o option.
		• The -I option specifying the include file path exceeds the limit. ⇒ Specify the -I option 8 times or less.
		• The command line ends with the -I option. ⇒ Specify the name of an include file after the –I option.
		• The string following the -D option is not of a character type (letter or underscore) that can be used in a macro name. Illegal macro name definition. ⇒ Specify the macro name correctly and define the macro correctly.
		• The command line ends with the -D option. ⇒ Specify a macro filename after the -D option.
		• The string following the -U option is not of a character type (letter or underscore) that can be used in a macro name. ⇒ Define the macro correctly.
		• You specified an illegal option on the cpp30 command line. ⇒ Specify only legal options.
11	cannot open input file.	• Input file not found. ⇒ Specify the correct input file name.
12	cannot close input file.	• Input file cannot be closed. ⇒ Check the input file name.
14	cannot open output file.	• Cannot open output file. ⇒ Specify the correct output file name.
15	cannot close output file.	• Cannot close output file. ⇒ Check the available space on disk.
16	cannot write output file	• Error writing to output file. ⇒ Check the available space on disk.

Table F.4 cpp30 Error Messages (2)

No.	Error message	Description and countermeasure
17	input file name buffer overflow	• The input filename buffer has overflowed. Note that the filename includes the path. ⇒ Reduce the length of the filename and path (use the -I option to specify the standard directory).
18	not enough memory for macro include file not found	• Insufficient memory for macro name and contents of macro ⇒ Increase the swap area
21	include file not found	• The include file could not be opened.. ⇒ The include files are in the current directory and that specified in the -I option and environment variable. Check these directories.
22	illegal file name error	• Illegal filename. ⇒ Specify a correct filename.
23	include file nesting over	• Nesting of include files exceeds the limit (40). ⇒ Reduce nesting of include files to a maximum of 8 levels.
25	illegal identifier	• Error in #define. ⇒ Code the source file correctly.
26	illegal operation	• Error in preprocess commands #if - #elseif - #assert operation expression. ⇒ Rewrite operation expression correctly.
27	macro argument error	• Error in number of macro parameters when expanding macro. ⇒ Check macro definition and reference and correct as necessary.
28	input buffer over flow	• Input line buffer overflow occurred when reading source file(s). Or, buffer overflowed when converting macros. ⇒ Reduce each line in the source file to a maximum of 1023 characters. If you anticipate macro conversion, modify the code so that no line exceeds 1023 characters after conversion.
29	EOF in comment	• End of file encountered in a comment. ⇒ Correct the source file.
31	EOF in preprocess command	• End of file encountered in a preprocess command ⇒ Correct the source file.
32	unknown preprocess command	• An unknown preprocess command has been specified. ⇒ Only the following preprocess commands can be used in CPP30 : #include, #define, #undef, #if, #ifdef, #ifndef, #else, #endif, #elseif, #line, #assert, #pragma, #error
33	new_line in string	• A new-line code was included in a character constant or character string constant. ⇒ Correct the program.
34	string literal out of range 509 characters	• A character string exceeded 509 characters. ⇒ Reduce the character string to 509 characters max.
35	macro replace nesting over	• Macro nesting exceeded the limit (20). ⇒ Reduce the nesting level to a maximum of 20.
41	include file error	• Error in #include instruction. ⇒ Correct.

Table F.5 cpp30 Error Messages (3)

No.	Error message	Description and countermeasure
43	illegal id name	• Error in following macro name or argument in #define command: __FILE__, __LINE__, __DATE__, __TIME__ ⇒ Correct the source file.
44	token buffer over flow	• Token character buffer of #define overflowed. ⇒ Reduce the number of token characters.
45	illegal undef command usage	• Error in #undef. ⇒ Correct the source file.
46	undef id not found	• The following macro names to be undefined in #undef were not defined: __FILE__, __LINE__, __DATE__, __TIME__ ⇒ Check the macro name.
52	illegal ifdef / ifndef command usage	• Error in #ifdef. ⇒ Correct the source file.
53	elseif / else sequence erro	• #elseif or #else were used without #if - #ifdef - #ifndef. ⇒ Use #elseif or #else only after #if - #ifdef -#ifndef.
54	endif not exist	• No #endif to match #if - #ifdef - #ifndef. ⇒ Add #endif to the source file.
55	endif sequence error	• #endif was used without #if - #ifdef - #ifndef. ⇒ Use #endif only after #if - #ifdef - #ifndef.
61	illegal line command usage	• Error in #line. ⇒ Correct the source file.

F.4 cpp30 Warning Messages

Table F.6 shows the warning messages output by cpp30 and their countermeasures.

Table F.6 cpp30 Warning Messages

No.	Warning Messages	Description and countermeasure
81	reserved id used	• You attempted to define or undefine one of the following macro names reserved by cpp30: __FILE__, __LINE__, __DATE__, __TIME__ ⇒ Use a different macro name.
82	assertion warning	• The result of an #assert operation expression was 0. Check the operation expression.
83	garbage argument	• Characters other than a comment exist after a preprocess command. ⇒ Specify characters as a comment (/* string */) after the preprocess command.
84	escape sequence out of range for character	• An escape sequence in a character constant or character string constant exceeded 255 characters. ⇒ Reduce the escape sequence to within 255 characters.
85	redefined	• A previously defined macro was redefined with different contents. ⇒ Check the contents against those in the previous definition.
87	/* within comment	• A comment includes /*. ⇒ Do not nest comments.
88	Environment variable 'NCKIN' must be 'SJIS' or 'EUC'	• Environment variable 'NCKIN' is not valid. ⇒ Set "SJIS" or "EUC" to NCKIN.
90	'Macro name' in #if is not defined,so it's tereated as 0	• An undefined macro name in #if is used. ⇒ Check the macro definition.

F.5 ccom30 Error Messages

Table F.7 to Table F.19 list the ccom30 compiler error messages and their countermeasures.

Table F.7 ccom30 Error Messages (1)

Error message	Description and countermeasure
#pragma PRAGMA-name functionname redefined	• The same function is defined twice in #pragmaname. ⇒ Make sure that #pragma-name is declared only once.
#pragma PRAGMA-name function argument is long-long or double	• The arguments used for the function specified with the "#pragma program name function name" are the long long type or the double type. ⇒ The long long type and double type cannot be used in the functions specified with the "#pragma program name function name." Use other types.
#pragma PRAGMA-name & function prototype mismatched	• The function specified by #pragma PRAGMAname does not match the contents of argument in prototype declaration. ⇒ Make sure it is matched to the argument in prototype declaration.
#pragma PRAGMA-name's function argument is struct or union	• The struct or union type is specified in the prototype declaration for the function specified by #pragma PRAGMA-name. ⇒ Specify the int or short type, 2-byte pointer type, or enumeration type in the prototype declaration.
#pragma PRAGMA-name must be declared before use	• A function specified in the #pragma PRAGMAname declaration is defined after call for that function. ⇒ Declare a function before calling it.
#pragma BITADDRESS variable is not _Bool type	• The variable spcified by #pragma BITADDRESS is not _Bool type ⇒ Use the _Bool type to declare the variable.
#pragma INTCALL function's argument on stack	• When the body of functions declared in #pragma INTCALL are written in C, the parameters are passed via the stack. ⇒ When the body of functions declared in #pragma INTCALL are written in C, specify the parameters are being passed via the stack.
#pragma PARAMETER function's register not allocated	• A register which is specifed in the function decleared by #pragma PARAMETER can not be allocated. ⇒ Use the correct register.
'const' is duplicate	• const is described more than twice. ⇒ Write the type qualifier correctly.
'far' & 'near' conflict	• far/near is described more than twice. ⇒ Write near/far correctly.
'far' is duplicate	• far is described more than twice. ⇒ Write far correctly.
'near' is duplicate	• near is described more than twice. ⇒ Write near correctly.
'static' is illegal storage class for agument	• An appropriate storage class is used in argument declaration. ⇒ Use the correct storage class.

Table F.8 ccom30 Error Messages (2)

Error message	Description and countermeasure
'volatile' is duplicate	• volatile is described more than twice. ⇒ Write the type qualifier correctly.
(can't read C source from filename line number for error message)	• The source line is in error and 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.
(can't open C source filename for error message)	• The source file in error cannot be opened. ⇒ Check whether the file exists.
argument type given both places	• Argument declaration in function definition overlaps an argument list separately given. ⇒ Choose the argument list or argument declaration for this argument declaration.
array of functions declared	• The array type in array declaration is defined as function. ⇒ Specify scalar type struct/union for the array type.
array size is not constant integer	• The number of elements in array declaration is not a constant. ⇒ Use a constant to describe the number of elements.
asm()'s string must have only 1 $b	• $b is described more than twice in asm statement. ⇒ Make sure that $b is described only once.
asm()'s string must not have more than 3 $$ or $@	• $$ or $@ is described more than thrice in asm statement. ⇒ Make sure that $$ ($@)is described only twice.
auto variable's size is zero	• An array with 0 elements or no elements was declared in the auto area. ⇒ Correct the coding.
bitfield width exceeded	• The bit-field width exceeds the bit width of the data type. ⇒ Make sure that the data type bit width declared in the bit-field is not exceeded.
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.
can't get bitfield address by '&' operator	• The bit-field type is written with the & operator. ⇒ Do not use the & operator to write the bit-field type.
can't get inline function's address by '&' operator	• The & operator is written in an inline function. ⇒ Do not use the & operator in an inline function.
can't get size of bitfield	• The bit-field type is written with the sizeof operator. ⇒ Do not use the sizeof operator to write the bitfield type.
can't get void value	• An attempt is made to get void-type data as in cases where the right side of an assignment expression is the void type. ⇒ Check the data type.
can't output to file-name	• The file cannot be wrote ⇒ Check the rest of disk capacity or access right of the file.
can't open file-name	• The file cannot be opened. ⇒ Check the permission of the file.
can't set argument	• The type of an actual argument does not match prototype declaration. The argument cannot be set in a register (argument). ⇒ Correct mismatch of the type.
can't refer to the range outside of the stack frame.	• A location outside the stack frame area is referenced. ⇒ Reference the correct location.

Table F.9 ccom30 Error Messages (3)

Error message	Description and countermeasure
case value is duplicated	• The value of case is used more than one time. ⇒ Make sure that the value of case that you used once is not used again within one switch statement.
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.
conflict function argument type of variable-name	• The argument list contains the same variable name. ⇒ Change the variable name.
declared register parameter function's body declared	• The function body for the function declared with #pragma PARAMETER is defined in C ⇒ Do not define , in C, the body for such function .
default function argument conflict	• The default value of an argument is declared more than once in prototype declaration. ⇒ Make sure that the default value of an argument is declared only once.
default: is duplicated	• The default value is used more than one time. ⇒ Use only one default within one switch statement.
do while(struct/union) statement	• The struct or union type is used in the expression of the do-while statement. ⇒ Use the scalar type for an expression in the dowhile statement.
do while(void) statement	• The void type is used in the expression of the dowhile statement. ⇒ Use the scalar type for an expression in the dowhile statement.
duplicate frame position defind variable-name	• Auto variable is described more than twice. ⇒ Write the type specifier correctly.
Empty declare	• Only storage class and type specifiers are found. ⇒ Write a declarator.
float and double not have sign	• Specifiers signed/unsigned are described in float or double. ⇒ Write the type specifier correctly.
floating point value overflow	• The floating-point immediate value exceeds the representable range. ⇒ Make sure the value is within the range.
floating type's bitfield	• A bit-field of an invalid type is declared. ⇒ Use the integer type to declare a bit-field.
for(; struct/union;) statement	• The struct or union type is used in the second expression of the for statement. ⇒ Use the scalar type to describe the second expression of the for statement.
for(; void ;) statement	• The 2nd expression of the for statement has void. ⇒ Use the scalar type as the 2nd expression of the for statement.
function initialized	• An initialize expression is described for function declaration. ⇒ Delete the initialize expression.
function member declared	• A member of struct or union is function type ⇒ Write the members correctly.

Table F.10 ccom30 Error message (4)

Error message	Description and countermeasure
function returning a function declared	• The type of the return value in function declaration is function type. ⇒ Change the type to "pointer to function"etc.
function returning an array	• The type of the return value in function declaration is an array type. ⇒ Change the type to "pointer to function"etc.
handler function called	• The function specified by #pragma HANDLER is called. ⇒ Be careful not to call a handler.
identifier (variable-name) is duplicated	• The variable is defined more than one time. ⇒ Specify variable definition correctly.
if(struct/union) statement	• The struct or union type is used in the expression of the if statement. ⇒ The expression must have scalar type.
if(void) statement	• The void type is used in the expression of the if statement. ⇒ The expression must have scalar type.
illegal storage class for argument, 'inline' ignored	• An inline function is declared in declaration statement within a function. ⇒ Declare it outside a function.
illegal storage class for argument, 'interrupt' ignored	• An interrupt function is declared in declaration statement within a function. ⇒ Declare it outside a function.
incomplete array access	• An attempt is made to reference an array of incomplete. ⇒ Define size of array.
incomplete return type	• An attempt is made to reference an return variable of incomplete type. ⇒ Check return variable.
incomplete struct get by []	• An attempt is made to reference or initialize an array of incomplete structs or unions that do not have defined members. ⇒ Define complete structs or unions first.
incomplete struct member	• An attempt is made to reference an struct member of incomplete . ⇒ Define complete structs or unions first.
incomplete struct initialized	• An attempt is made to initialize an array of incomplete structs or unions that do not have defined members. ⇒ Define complete structs or unions first.
incomplete struct return function call	• An attempt is made to call a function that has as a return value the of incomplete struct or union that does not have defined members. ⇒ Define a complete struct or union first.
incomplete struct / union's member access	• An attempt is made to reference members of an incomplete struct or union that do not have defined members. ⇒ Define a complete struct or union first.
incomplete struct / union(tagname)' s member access	• An attempt is made to reference members of an incomplete struct or union that do not have defined members. ⇒ Define a complete struct or union first.
inline function have invalid argument or return code	• inline function has an invalid argument or an invalid return value. ⇒ Write the argument or an invalid return value correctly.

Table F.11 ccom30 Error message (5)

Error message	Description and countermeasure
inline function is called as normal function before	• The function declared in storage class inline is called as an ordinary function. ⇒ Always be sure to define an inline function before using it.
inline function's address used	• An attempt is made to reference the address of an inline function. ⇒ Do not use the address of an inline function.
inline function's body is not declared previously	• The body of an inline function is not defined. ⇒ Using an inline function, define the function body prior to the function call.
inline function (function-name) is recursion	• The recursive call of an in line function cannot be carried out. ⇒ Using an inline function, No recursive.
interrupt function called	• The function specified by #pragma INTERRUPT is called. ⇒ Be careful not to call an interrupt handling function.
invalid environment variable: (environment variable -name)	• The variable name specified in the environment variable NCKIN/NCKOUT is specified by other than SJIS and EUC. ⇒ Check the environment variables used.
invalid function default argument	• The default argument to the function is incorrect. ⇒ This error occurs when the prototype declaration of the function with default arguments and those in the function definition section do not match. Make sure they match.
invalid push	• An attempt is made to push void type in function argument, etc. ⇒ The type void cannot be pushed.
invalid ' ? : ' operand	• The ?: operation contains an error. ⇒ Check each expression. Also note that the expressions on the left and right sides of : must be of the same type.
invalid '!=' operands	• The != operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '&&' operands	• The && operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '&' operands	• The & operation contains an error. ⇒ Check the expression on the right side of the operator.
invalid '&=' operands	• The &= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '()' operand	• The expression on the left side of ( ) is not a function. ⇒ Write a function or a pointer to the function in the left-side expression of ( ).
invalid '*' operands	• If multiplication, the * operation contains an error. If * is the pointer operator, the right-side expressionis 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.

Table F.12 ccom30 Error message (6)

Error message	Description and countermeasure
invalid '*=' operands	• The *= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '+' operands	• The + operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '+=' operands	• The += operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '-' operands	• The - operator contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '-=' operands	• The -= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '/=' operands	• The /= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '<<' operands	• The << operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '<<=' operands	• The <<= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '<=' operands	• The <= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '=' operand	• The = operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '= =' operands	• The = = operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '>=' operands	• The >= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '>>' operands	• The >> operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '>>=' operands	• The >>= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '[ ]' operands	• The left-side expression of [ ] is not array type or pointer type. ⇒ Use an array or pointer type to write the left-side expression of [ ].

Table F.13 ccom30 Error message (7)

Error message	Description and countermeasure
invalid '^=' operands	• The ^= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '|=' operands	• The | = operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '||' operands	• The | | operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '%=' operands	• The %= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid ++ operands	• The ++ unary operator or postfix operator contains an error. ⇒ For the unary operator, check the right-side expression. For the postfix operator, check the leftside expression.
invalid -- operands	• The -- unary operation or postfix operation contains an error. ⇒ For the unary operator, check the right-side expression. For the postfix operator, check the leftside expression.
invalid -> used	• The left-side expression of -> is not struct or union. ⇒ The left-side expression of -> must have struct or union.
invalid (? :)'s condition	• The ternary operator is erroneously written. ⇒ Check the ternary operator.
invalid array type	• Incomplete arrays cannot be declared. ⇒ Specify the number of elements in the multidimensional array.
invalid operation for pointer to incomplete type	• Invalid calculation for the pointer to an incomplete type. ⇒ Define members of a structure or define complete structs.
Invalid #pragma OS Extended function interrupt number	• The INT No. in #pragma OS Extended function is invalid. ⇒ Specify correctly.
Invalid #pragma INTCALL interrupt number	• The INT No. in #pragma INTCALL is invalid. ⇒ Specify correctly.
Invalid #pragma SPECIAL special page number	• The number or format specification written with #pragma SPECIAL is incorrect. ⇒ Specify the number or format correctly.
Invalid #pragma INTERRUPT vector number	• The number or format specification written with #pragma INTERRUPT is incorrect. ⇒ Specify the number or format correctly.
invalid CAST operand	• The cast operation contains an error. The void type cannot be cast to any other type; it can neither be cast from the structure or union type nor can it be cast to the structure or union type. ⇒ Write the expression correctly.
invalid asm()'s argument	• The variables that can be used in asm statements are only the auto variable and argument. ⇒ Use the auto variable or argument for the statement.
invalid bitfield declare	• The bit-field declaration contains an error. ⇒ Write the declaration correctly.
invalid break statements	• The break statement is put where it cannot be used. ⇒ Make sure that it is written in switch, while, dowhile, and for.

Table F.14 ccom30 Error message (8)

Error message	Description and countermeasure
invalid case statements	• The switch statement contains an error. ⇒ Write the switch statement correctly.
invalid case value	• The case value contains an error. ⇒ Write an integral-type or enumerated-type constant.
invalid cast operator	• Use of the cast operator is illegal. ⇒ Write the expression correctly.
invalid continue statements	• The continue statement is put where it cannot be used. ⇒ Use it in a while, do-while, and for block.
invalid default statements	• The switch statement contains an error. ⇒ Write the switch statement correctly.
invalid enumerator initialized	• The initial value of the enumerator is incorrectly specified by writing a variable name, for example. ⇒ Write the initial value of the enumerator correctly.
invalid function argument	• An argument which is not included in the argument list is declared in argument definition in function definition. ⇒ Declare arguments which are included in the argument list.
invalid function's argument declaration	• The argument of the function is erroneously declared. ⇒ Write it correctly.
invalid function declare	• The function definition contains an error. ⇒ Check the line in error or the immediately preceding function definition.
invalid initializer	• The initialization expression contains an error. This error includes excessive parentheses, many initialize expressions, a static variable in the function initialized by an auto variable, or a variable initialized by another variable. ⇒ Write the initialization expression correctly.
invalid initializer of variable-name	• The initialization expression contains an error. This error includes a bit-field initialize expression described with variables, for example. ⇒ Write the initialization expression correctly.
invalid initializer on array	• The initialization expression contains an error. ⇒ Check to see if the number of initialize expressions in the parentheses matches the number of array elements and the number of structure members.
invalid initializer on char array	• The initialization expression contains an error. ⇒ Check to see if the number of initialize expressions in the parentheses matches the number of array elements and the number of structure members.
invalid initializer on scalar	• The initialization expression contains an error. ⇒ Check to see if the number of initialize expressions in the parentheses matches the number of array elements and the number of structure members.
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.
invalid initializer, too many brace	• Too many braces { } are used in a scalar-type initialization expression of the auto storage class. ⇒ Reduce the number of braces { } used.

Table F.15 ccom30 Error message (9)

Error message	Description and countermeasure
invalid lvalue	• The left side of the assignment statement is not lvalue. ⇒ Write a substitutable expression on the left side of the statement.
invalid lvalue at '=' operator	• The left side of the assignment statement is not lvalue. ⇒ Write a substitutable expression on the left side of the statement.
invalid member	• The member reference contains an error. ⇒ Write correctly.
invalid member used	• The member reference contains an error. ⇒ Write correctly.
invalid redefined type name of (identifier)	• The same identifier is defined more than once in typedef. ⇒ Write the identifier correctly.
invalid return type	• The type of return value of the function is incorrect. ⇒ Write it correctly.
invalid sign specifier	• Specifiers signed/unsigned are described twice or more. ⇒ Write the type specifier correctly.
invalid storage class for data	• The storage class is erroneously specified. ⇒ Write it correctly.
invalid struct or union type	• Structure or union members are referenced for the enumerated type of data. ⇒ Write it correctly.
invalid truth expression	• The void, struct, or union type is used in the first expression of a condition expression (?:). ⇒ Use scalar type to write this expression.
invalid type specifier	• The same type specifier is described twice or more as in "int int i;" or an incompatible type specifier is described as in "float int i;." ⇒ Write the type specifier correctly.
invalid type's bitfield	• A bit-field of an invalid type is declared. ⇒ Use the integer type for bit-fields.
invalid type specifier,long long long	• Specifiers “long” are described thrice or more. ⇒ Check the type.
invalid unary '!' operands	• Use of the ! unary operator is illegal. ⇒ Check the right-side expression of the operator.
invalid unary '+' operands	• Use of the + unary operator is illegal. ⇒ Check the right-side expression of the operator.
invalid unary '-' operands	• Use of the - unary operator is illegal. ⇒ Check the right-side expression of the operator.
invalid unary '~' operands	• Use of the ~ unary operator is illegal. ⇒ Check the right-side expression of the operator.
invalid void type	• The void type specifier is used with long or singed. ⇒ Write the type specifier correctly.
invalid void type, int assumed	• The void-type variable cannot be declared. Processing will be continued by assuming it to be the int type. ⇒ Write the type specifier correctly.
invalid size of bitfield	• Get the bitfield size. ⇒ Not write bitfield on this decraration.

Table F.16 ccom30 Error message (10)

Error message	Description and countermeasure
invalid switch statement	• The switch statement is illegal. ⇒ Write it correctly.
label label redefine	• The same label is defined twice within one function. ⇒ Change the name for either of the two labels.
long long type's bitfield	• Specifies bitfield by long long type ⇒ Can not specifies bit-fields of long long type.
mismatch prototyped parameter type	• The argument type is not the type declared in prototype declaration. ⇒ Check the argument type.
No #pragma ENDASM	• #pragma ASM does not have matching #pragma ENDASM. ⇒ Write #pragma ENDASM.
No declarator	• The declaration statement is incomplete. ⇒ Write a complete declaration statement.
Not enough memory	• The memory area is insufficient. ⇒ Increase the memory or virtual memory for Windows.
not have 'long char'	• Type specifiers long and char are simultaneously used. ⇒ Write the type specifier correctly.
not have 'long float'	• Type specifiers long and float are simultaneously used. ⇒ Write the type specifier correctly.
not have 'long short'	• Type specifiers long and short are simultaneously used. ⇒ Write the type specifier correctly.
not static initializer for variablename	• The initialize expression of static variable contains an error. This is because the initialize expression is a function call, for example. ⇒ Write the initialize expression correctly.
not struct or union type	• The left-side expression of -> is not the structure or union type. ⇒ Use the structure or union type to describe the left-side expression of ->.
redeclare of variable-name	• An variable-name has been declared twice. ⇒ Change the name for either of the two variable name.
redeclare of enumerator	• An enumerator has been declared twice. ⇒ Change the name for either of the two enumerators.
redefine function function-name	• The function indicated by function-name is defined twice. ⇒ The function can be defined only once. Change the name for either of the two functions.
redefinition tag of enum tag-name	• An enumeration is defined twice. ⇒ Make sure that enumeration is defined only once.
redefinition tag of struct tag-name	• A structure is defined twice. ⇒ Make sure that a structure is defined only once.
redefinition tag of union tag-name	• A union is defined twice. ⇒ Make sure that a union is defined only once.
reinitialized of variable-name	• An initialize expression is specified twice for the same variable. ⇒ Specify the initializer only once.

Table F.17 ccom30 Error message (11)

Error message	Description and countermeasure
restrict is duplicate	• A restrict is defined twice. ⇒ Make sure that a restrict is defined only once.
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.
size of incomplete type	• An undefined structure or union is used in the operand of the sizeof operator. ⇒ Define the structure or union first. • The number of elements of an array defined as an operand of the sizeof operator is unknown. ⇒ Define the structure or union first.
size of void	• An attempt is made to find the size of void. This is an invalid size. ⇒ The size of void cannot be found.
Sorry, stack frame memory exhaust, max. 64(or 255) bytes but now nnn bytes	• A maximum of 128 bytes of parameters can be secured on the stack frame. Currently, nnn bytes have been used. ⇒ Reduce the size or number of parameters.
Sorry, compilation terminated because of these errors in functionname.	• An error occurred in some function indicated by function-name. Compilation is terminated. ⇒ Correct the errors detected before this message is output.
Sorry, compilation terminated because of too many errors.	• Errors in the source file exceeded the upper limit (50 errors). ⇒ Correct the errors detected before this message is output.
struct or enum's tag used for union	• The tag name for structure and enumerated type is used as a tag name for union. ⇒ Change the tag name.
struct or union's tag used for enum	• The tag name for structure and union is used as a tag name for enumerated type. ⇒ Change the tag name.
struct or union,enum does not have long or sign	• Type specifiers long or signed are used for the struct/union/enum type specifiers. ⇒ Write the type specifier correctly.
switch's condition is floating	• The float type is used for the expression of a switch statement. ⇒ Use the integer type or enumerated type.
switch's condition is void	• The void type is used for the expression of a switch statement. ⇒ Use the integer type or enumerated type.
switch's condition must integer	• Invalid types other than the integer and enumerated types are used for the expression of a switch statement. ⇒ Use the integer type or enumerated type.
syntax error	• This is a syntax error. ⇒ Write the description correctly.

Table F.18 ccom30 Error message (12)

Error message	Description and countermeasure
System Error	• This is an internal error. (It does not normally occur.) This error may occur pursuant to one of errors that occurred before it. ⇒ If this error occurs even after eliminating all errors that occurred before it, please send the content of the error message to Renesas Solutions Corp. as you contact.
too big data-length	• An attempt is made to get an address exceeding the 32-bit range. ⇒ Make sure the set values are within the address range of the microcomputer used.
too big address	• An attempt is made to set an address exceeding the 32-bit range. ⇒ Make sure the set values are within the address range of the microcomputer used.
too many storage class of typedef	• Storage class specifiers such as extern/typedef/ static/auto/register are described more than twice in declaration. ⇒ Do not describe a storage class specifier more than twice.
type redeclaration of variable-name	• The variable is defined with different types each time. ⇒ Always use the same type when declaring a variable twice.
typedef initialized	• An initialize expression is described in the variable declared with typedef. ⇒ Delete the initialize expression.
uncomplete array pointer operation	• An incomplete multidimensional array has been accessed to pointer. ⇒ Specify the size of the multidimensional array.
undefined label "label" used	• The jump-address label for goto is not defined in the function. ⇒ Define the jump-address label in the function.
union or enum's tag used for struct	• The tag name for union and enumerated types is used as a tag name for structure. ⇒ Change the tag name.
unknown function argument variable-name	• An argument is specified that is not included in the argument list. ⇒ Check the argument.
unknown member "member-name" used	• A member is referenced that is not registered as any structure or union members. ⇒ Check the member name.
unknown pointer to structure identifier"variable-name"	• The left-side expression of -> is not the structure or union type. ⇒ Use struct or union as the left-side expression of ->.
unknown size of struct or union	• A structure or union is used which has had its size not determined. ⇒ Declare the structure or union before declaring a structure or union variable.

Table F.19 ccom30 Error message (13)

Error message	Description and countermeasure
unknown structure identifier "variable-name"	• The left-side expression of "." dose not have struct or union. ⇒ Use the struct or union as it.
unknown variable "variable-name" used in asm()	• An undefined variable name is used in the asm statement. ⇒ Define the variable.
unknown variable variable-name	• An undefined variable name is used. ⇒ Define the variable.
unknown variable variable-name used	• An undefined variable name is used. ⇒ Define the variable.
void array is invalid type, int array assumed	• An array cannot be declared as void. Processing will be continued, assuming it has type int. ⇒ Write the type specifier correctly.
void value can't return	• The value converted to void (by cast) is used as the return from a function. ⇒ Write correctly.
while(struct/union) statement	• struct or union is used in the expression of a while statement. ⇒ Use scalar type.
while(void) statement	• void is used in the expression of a while statement. ⇒ Use scalar type.
multiple #pragma EXT4MPTR's pointer, ignored (NC30 only)	• # pragma EXT4MPTR is declared more than two. ⇒ Do not declare #pragma EXT4MPTR more than two.
zero size array member	• the array which size is zero. ⇒ Declare the array size. • The structure members include an array whose size is zero. ⇒ Arrays whose size is zero cannot be members of a structure.
'function-name' is resursion, then inline is ignored	• The inline-declared 'function name' is called recursively. The inline declaration will be ignored. ⇒ Correct the statement not to call such a function name recursively.

F.6 c ccom30 Warning Messages

Table F.20 to Table F.28 list the ccom30 compiler warning messages and their countermeasures.

Table F.20 ccom30 Warning Messages (1)

Warning message	Description and countermeasure
#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.
#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.
#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.
#pragma pragma-name format error	• The #pragma pragma-name is erroneously written. Processing will be continued. ⇒ Write it correctly.
#pragma pragma-name format error, ignored	• The #pragma pragma-name is erroneously written. This line will be ignored. ⇒ Write it correctly.
#pragma pragma-name not function, ignored	• A name is written in the #pragma pragma-name that is not a function. ⇒ Write it with a function name.
#pragma pragma-name's function must be predeclared, ignored	• A function specified in the #pragma pragma-name is not declared. ⇒ For functions specified in a #pragma pragmaname, write prototype declaration in advance.
#pragma pragma-name's function must be prototyped, ignored	• A function specified in the #pragma pragma-name is not prototype declared. ⇒ For functions specified in a #pragma pragmaname, write prototype declaration in advance.
#pragma pragma-name's function return type invalid,ignored	• The type of return value for a function specified in the #pragma pragma-name is invalid. ⇒ Make sure the type of return value is any type other than struct, union, or double.
#pragma pragma-name unknown switch,ignored	• The switch specified in the #pragma pragma-name is invalid. ⇒ Write it correctly.
#pragma pragma-name variable initialized, initialization ignored	• The variable specified in #pragma pragma-name is initialized. The specification of #pragma pragma-name will be nullified. ⇒ Delete either #pragma pragma-name or the initialize expression.
#pragma ASM line too long, then cut	• The line in which #pragma ASM is written exceeds the allowable number of characters = 1,024 bytes. ⇒ Write it within 1,024 bytes.

Table F.21 ccom30 Warning Messages (2)

Warning message	Description and countermeasure
#pragma directive conflict	• #pragma of different functions is specified for one function. ⇒ Write it correctly.
#pragma DMAC duplicate (only NC308)	• The same #pragma DMAC is defined twice. ⇒ Do not define #pragma DMAC two times or more
#pragma DMAC variable must be far pointer for variable-name, ignored (only NC308)	• Variable declared by #pragma DMAC needs to be a far pointer.DMAC declaration is ignored. ⇒ Write it correctly.
#pragma DMAC variable must be unsigned int for variable-name, ignored (only NC308)	• Variable declared by #pragma DMAC needs to be unsigned int type.DMAC declaration is ignored. ⇒ Write it correctly.
#pragma DMAC's variable must be pre-declared,ignored (only NC308)	• Variable declared by #pragma DMAC needs a type declaration. ⇒ Write it correctly.
#pragma DMAC, register conflict (only NC308)	• Multiple variables are allocated to the same register. ⇒ Write it correctly.
#pragma DMAC, unknown register name used (only NC308)	• Unknown register is used in #pragma DMAC declaration. ⇒ Write it correctly.
#pragma JSRA illegal location, ignored	⇒ Do not put #pragma JSRA inside function scope. • Write #pragma JSRA outside a function.
#pragma JSRW illegal location, ignored	• Do not put #pragma JSRW inside function scope. ⇒ Write #pragma JSRA outside a function.
#pragma PARAMETER function's address used	• The address of the function specified by #pragma PARAMETER is referenced. ⇒ Do not reference that address.
#pragma control for function duplicate, ignored	• Two or more of INTERRUPT, TASK, HANDLER, CYCHANDLER, or ALMHANDLER are specified for the same function in #pragma. ⇒ Be sure to specify only one of INTERRUPT, T A S K , H A N D L E R , C Y C H A N D L E R , o r ALMHANDLER.
#pragma unknown switch, ignored	• Invalid switch is specified to #pragma.#pragma declaration is ignored. ⇒ Write switch correctly.
'auto' is illegal storage class	• An incorrect storage class is used. ⇒ Specify the correct storage class.
'register' is illegal storage class	• An incorrect storage class is used. ⇒ Specify the correct storage class.
argument is define by 'typedef', 'typedef' ignored	• Specifier typedef is used in argument declaration. Specifier typedef will be ignored. ⇒ Delete typedef.
assign far pointer to near pointer, bank value ignored	• The bank address will be nullified when substituting the far pointer for the near pointer. ⇒ Check the data types, near or far.
assignment from const pointer to non-const pointer	• The const property is lost by assignment from const pointer to non-const pointer. ⇒ Check the statement description. If the description is correct, ignore this warning.

Table F.22 ccom30 Warning Messages (3)

Warning message	Description and countermeasure
assignment from volatile pointer to non-volatile pointer	• The volatile property is lost by assignment from volatile pointer to non-volatile pointer. ⇒ Check the statement description. If the description is correct, ignore this warning.
assignment in comparison statement	• You put an assignment expression in a comparison statement. ⇒ You may confuse "= =" with '='. Check on it.
block level extern variable initialize forbid,ignored	• An initializer is written in extern variable declaration in a function. ⇒ Delete the initializer or change the storage class.
can't get address from register storage class variable	• The & operator is written for a variable of the storage class register. ⇒ Do not use the & operator to describe a variable of the storage class register.
can't get size of bitfield	• The bit-field is used for the operand of the sizeof operator. ⇒ Write the operand correctly.
can't get size of function	• A function name is used for the operand of the sizeof operator. ⇒ Write the operand correctly.
can't get size of function, unit size 1 assumed	• The pointer to the function is incremented (++) or decremented (--). Processing will be continued by assuming the increment or decrement value is 1. ⇒ Do not increment (++) or decrement (--) the pointer to a function.
char array initialized by wchar_t string	• The array of type char is initialized with type wchar_t . ⇒ Make sure that the types of initializer are matched.
case value is out of range	• The value of case exceeds the switch parameter range. ⇒ Specify correctly.
character buffer overflow	• The size of the string exceeded 512 characters. ⇒ Do not use more than 512 characters for a string.
character constant too long	• There are too many characters in a character constant (characters enclosed with single quotes). ⇒ Write it correctly.
constant variable assignment	• In this assign statement, substitution is made for a variable specified by the const qualifier. ⇒ Check the declaration part to be substituted for.
cyclic or alarm handler function has argument	• The function specified by #pragma CYCHANDLER or ALMHANDLER is using an argument. ⇒ The function cannot use an argument. Delete the argument.
enumerator value overflow size of unsigned char	• The enumerator value exceeded 255. ⇒ Do not use more than 255 for the enumerator; otherwise, do not specify the startup function - fchar_enumerator.
enumerator value overflow size of unsigned int	• The enumerator value exceeded 65535. ⇒ Do not use more than 65535 to describe the enumerator.
enum's bitfield	• An enumeration is used as a bit field member. ⇒ Use a different type of member.
external variable initialized,change to public	• An initialization expression is specified for an extern-declared variable. extern will be ignored. ⇒ Delete extern.

Table F.23 ccom30 Warning Messages (4)

Warning message	Description and countermeasure
far pointer (implicitly) casted by near pointer	• The far pointer was converted into the near pointer. ⇒ Check the data types, near or far.
function must be far	• The function is declared with the near type. ⇒ Write it correctly.
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.
handler function called	• The function specified by #pragma HANDLER is called. ⇒ Be careful not to call a handler.
handler function can't return value	• The function specified by #pragma HANDLER is using a returned value. ⇒ The function specified by #pragma HANDLER cannot use a returned value. Delete the return value.
handler function has argument	• The function specified by #pragma HANDLER is using an argument. ⇒ The function specified by #pragma HANDLER cannot use an argument. Delete the argument.
hex character is out of range	• The hex character in a character constant is excessively long. Also, some character that is not a hex representation is included after \. ⇒ Reduce the length of the hex character.
identifier (member-name) is duplicated, this declare ignored	• The member name is defined twice or more. This declaration will be ignored. ⇒ Make sure that member names are declared only once.
identifier (variable-name) is duplicated	• The variable name is defined twice or more. This declaration will be ignored. ⇒ Make sure that variable names are declared only once.
identifier (variable-name) is shadowed	• The auto variable which is the same as the name declared as an argument is used. ⇒ Use any name not in use for arguments.
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.
incomplete array access	• An incomplete multidimensional array has been accessed. ⇒ Specify the size of the multidimensional array.
incompatible pointer types	• The object type pointed to by the pointer is incorrect. ⇒ Check the pointer type.
incomplete return type	• An attempt is made to reference an return variable of incomplete type. ⇒ Check return variable.
incomplete struct member	• An attempt is made to reference an struct member of incomplete . ⇒ Define complete structs or unions first.
init elements overflow,ignored	• The initialization expression exceeded the size of the variable to be initialized. ⇒ Make sure that the number of initialize expressions does not exceed the size of the variables to be initialized.
inline function is called as normal function before, change to static function	• The function declared in storage class inline is called as an ordinary function. ⇒ Always be sure to define an inline function before using it.

Table F.24 ccom30 Warning Messages (5)

Warning message	Description and countermeasure
integer constant is out of range	• The value of the integer constant exceeded the value that can be expressed by unsigned long. ⇒ Use a value that can be expressed by unsigned long to describe the constant.
interrupt function called	• The function specified by #pragma INTERRUPT is called. ⇒ Be careful not to call an interrupt handling function.
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 function. Delete the return value.
interrupt function has argument	• The interrupt handling function specified by #pragma INTERRUPT is using an argument. ⇒ Arguments cannot be used in an interrupt function. Delete the argument.
invalid #pragma EQU	• The description of #pragma EQU contains an error. This line will be ignored. ⇒ Write the description correctly.
invalid #pragma SECTION, unknown section base name	• The section name in #pragma SECTION contains an error. The section names that can be specified are data, bss, program, rom, interrupt, and bas. This line will be ignored. ⇒ Write the description correctly.
invalid #pragma operand, ignored	• An operand of #pragma contains an error. This line will be ignored. ⇒ Write the description correctly.
invalid function argument	⇒ The function argument is not correctly written. • Write the function argument correctly.
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.
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.
Kanji in #pragma ADDRESS	• The line of #pragma ADDRESS contains kanji code. This line will be ignored. ⇒ Do not use kanji code in this declaration.
Kanji in #pragma BITADDRESS	• The line of #pragma BITADDRESS contains kanji code. This line will be ignored. ⇒ Do not use kanji code in this declaration.
keyword (keyword) are reserved for future	• A reversed keyword is used. ⇒ Change it to a different name.
large type was implicitly cast to small type	• The upper bytes (word) of the value may be lost by assignment from large type to a smaller type. ⇒ Check the type. If the description is correct, ignore this warning.
mismatch prototyped parameter type	• The argument type is not the type declared in prototype declaration. ⇒ Check the argument type.

Table F.25 ccom30 Warning Messages (6)

Warning message	Description and countermeasure
meaningless statements deleted in optimize phase	• Meaningless statements were deleted during optimization. ⇒ Delete meaningless statements.
meaningless statement	• The tail of a statement is "= =". ⇒ You may confuse "=" with '= ='. Check on it.
mismatch function pointer assignment	• The address of a function having a register argument is substituted for a pointer to a function that does not have a register argument (i.e., a nonprototyped function). ⇒ Change the declaration of a pointer variable for function to a prototype declaration.
multi-character character constant	• A character constant consisting of two characters or more is used. ⇒ Use a wide character (L'xx') when two or more characters are required.
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.
No hex digit	• The hex constant contains some character that cannot be used in hex notation. ⇒ Use numerals 0 to 9 and alphabets A to F and a to f to describe hex constants.
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 appropriate value.
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 handled as int when processed. ⇒ Write the storage-class and type specifiers.
non-initialized variable "variable name" is used	• It is probable that uninitialized variables are being referenced. ⇒ Check the statement description. This warning can occur in the last line of the function. In such a case, check the description of the auto variables, etc. in the function. If the description is correct, ignore this warning.
non-prototyped function used	• A function is called that is not declared of the prototype. This message is output only when you specified the -Wnon_prototype option. ⇒ Write prototype declaration. Or delete the option "-Wnon_prototype".
non-prototyped function declared	• A prototype declaration for the defined function cannot be found. (Displayed only when the -Wnon_prototype option is specified.) ⇒ Write a prototype declaration.
octal constant is out of range	• The octal constant contains some character that cannot be used in octal notation. ⇒ Use numerals 0 to 7 to describe octal constants.
octal_character is out of range	• The octal constant contains some character that cannot be used in octal notation. ⇒ Use numerals 0 to 7 to describe octal constants.

Table F.26 com30 Warning Messages (7)

Warning message	Description and countermeasure
overflow in floating value converting to integer	• A very large floating-point number that cannot be stored in integer type is being assigned to the integer type. ⇒ Reexamine the assignment expression.
old style function declaration	• The function definition is written in format prior to ANSI (ISO) C. ⇒ Write the function definition in ANSI (ISO) format.
prototype function is defined as non-prototype function before.	• The non-prototyped function is redefine prototype-declaration. ⇒ Unite ways to declare function type.
redefined type	• Redwfine typedef. ⇒ Check typedef.
redefined type name of (qualify)	• The same identifier is defined twice or more in typedef. ⇒ Write identifier correctly.
register parameter function used before as stack parameter function	• The function for register argument is used as a function for stack argument before. ⇒ Write a prototype declaration before using the function.
RESTRICT qualifier can set only pointer type.	• The RESTRICT qualifier is declared outside a pointer. ⇒ Declare it in only a pointer.
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.
size of incomplete type	• An undefined structure or union is used in the operand of the size of operator. ⇒ Define the structure or union first.
	• The number of elements of an array defined as an operand of the size of operator is unknown. ⇒ Define the structure or union first.
size of incomplete array type	• An attempt is made to find size of of an array of unknown size. This is an invalid size. ⇒ Specify the size of the array.
size of void	• An attempt is made to find the size of void. This is an invalid size. ⇒ The size of void cannot be found.
standard library "function-name()" need "include-file name"	• This standard library function is used without its header file included. ⇒ Be sure to include the header file.
static variable in inline function	• static data is declared within a function that is declared in storage class inline. ⇒ Do not declare static data in an inline function.
string size bigger than array size	• The size of the initialize expression is greater than that of the variable to be initialized. ⇒ Make sure that the size of the initialize expression is equal to or smaller than the variable.
string terminator not added	• Since the variable to be initialized and the size of the initialize expression are equal, '\0' cannot be affixed to the character string. ⇒ Increase a element number of array.

Table F.27 ccom30 Warning Messages (8)

Warning message	Description and countermeasure
struct (or union) member's address can't has no near far information	• near or far is used as arrangement position information of members (variables) of a struct (or union). ⇒ Do not specify near and far for members.
task function called	• The function specified by #pragma TASK is called. ⇒ Be careful not to call a task function.
task function can't return value	• The function specified by #pragma TASK is using a return value. ⇒ The function specified by #pragma TASK cannot use return values. Delete the return value.
task function has invalid argument	• The function specified with #pragma TASK uses arguments. ⇒ Any function specified with #pragma TASK cannot use arguments. Delete the arguments.
this comparison is always false	• Comparison is made that always results in false. ⇒ Check the conditional expression.
this comparison is always true	• Comparison is made that always results in true. ⇒ Check the conditional expression.
this feature not supported now, ignored	• This is a syntax error. Do not this syntax because t is reserved for extended use in the future. ⇒ Write the description correctly.
this function used before with non-default argument	• A function once used is declared as a function hat has a default argument. ⇒ Declare the default argument before using a unction.
this interrupt function is called as normal function before	• A function once used is declared in #pragma NTERRUPT. ⇒ An interrupt function cannot be called. Check the ontent of #pragma.
too big octal character	• The character constant or the octal constant in he character string exceeded the limit value (255 n decimal). ⇒ Do not use a value greater than 255 to describe he constant.
too few parameters	• Arguments are insufficient compared to the number f arguments declared in prototype declaration. ⇒ Check the number of arguments.
too many parameters	• Arguments are excessive compared to the number f arguments declared in prototype declaration. ⇒ Check the number of arguments.
unknown #pragma STRUCT xxx	• #pragma STRUCTxxx cannot be processed. his line will be ignored. ⇒ Write correctly.
Unknown debug option (-dx)	• The option -dx cannot be specified. ⇒ Specify the option correctly.
Unknown function option (-Wxxx)	• The option -Wxxx cannot be specified. ⇒ Specify the option correctly.
Unknown function option (-fx)	• The option -fx cannot be specified. ⇒ Specify the option correctly.
Unknown function option (-gx)	• The option -gx cannot be specified. ⇒ Specify the option correctly.

Table F.28 ccom30 Warning Messages (9)

Warning message	Description and countermeasure
Unknown optimize option (-mx)	• The option -mx cannot be specified. ⇒ Specify the option correctly.
Unknown optimize option (-Ox)	• The option -Ox cannot be specified. ⇒ Specify the option correctly.
Unknown option (-x)	• The option -x cannot be specified. ⇒ Specify the option correctly.
unknown pragma pragma-specification used	• Unsupported #pragma is written. ⇒ Check the content of #pragma. *This warning is displayed only when the Wunknown_pragma (-WUP) option is specified.
wchar_t array initialized by char string	• The initialize expression of the wchar_t type is nitialized by a character string of the char type. ⇒ Make sure that the types of the initialize expression re matched.
zero divide in constant folding	• The divisor in the divide operator or remainder alculation operator is 0. ⇒ Use any value other than 0 for the divisor.
zero divide,ignored	• The divisor in the divide operator or remainder alculation operator is 0. ⇒ Use any value other than 0 for the divisor.
zero width for bitfield	• The bit-field width is 0. ⇒ Write a bit-field equal to or greater than 1.
no const in previous declaretion	• The function or variable declaration without const qualification is const-qualified on the entity definition side. ⇒ Make sure the function or variable declaration and the const qualification on the entity definition side are matched.
xxx was declared but never referenced	• There is a declaration that is not referenced. ⇒ Delete the declaration.

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

The SBDATA declaration & SPECIAL page Function declaration Utility utl30 precesses the absolute module file (hanving the extension.x30).
The utl30 generates a file that contains SBDATA declarations (located in the SB area beginning with the most frequently used one,"#pragma SBDATA") and a file that contains SPECIAL page function declarations (located in the SPECIAL page area beginning with the most frequently used one,"#pragma SPECIAL").
To use utl30, specify the compile driver startup option -finfo when compiling, so that the absolute module file (.x30) will be generated.
Figure G.1 illustrates the NC30 processing flow.

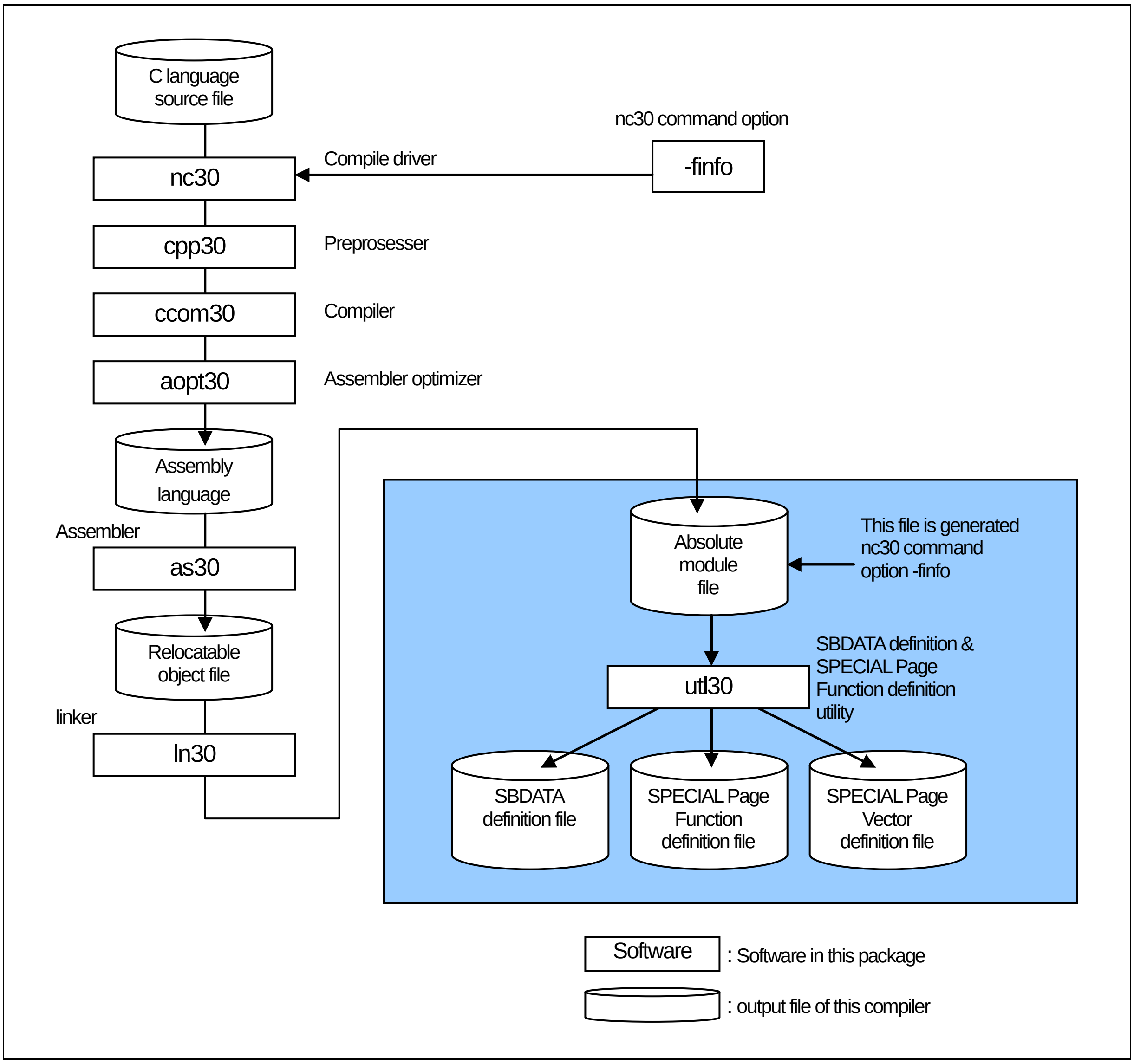


Figure G.1 NC30 Processing Flow

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△[command-line-option]. <absolute-file-name>

%: Prompt
< >: Mandatory item
[ ]: Optional item
△: Space
Delimit multiple command line options with spaces.
```

Figure G.2 utl30 Command Line Format

Before utl30 can be used, the following startup options of the compiler must both be specified in order to generate an absolute module file (extension .x30):

- -finfo option to output an inspector information
- -g option to output debugging information

The following utl30 options are also specified:

- -o option to output of information(SBDATA declaration or SPECIAL page Function declaration)

(By default, information is output to the standard output device.)

● Output the absolute module file

```
%nc30 ncrt0.a30 -finfo sample.c<RET>
M16C Series, R8C Family Compiler V.x.xx Release xx
Copyright(C) xxxx(xxxx). Renesas Electronics Corp.
and Renesas Solutions Corp., All rights reserved.
ncrt0.a30
sample.c

%
```

● Output SBDATA declaration

```
%utl30 -sb30 ncrt0.x30 -o sample<RET>
M16C/60 UTILITY UTL30 for M16C/60 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS ELECTRONICS CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%
```

● Output SPECIAL page Function declaration

```
%utl30 -sp30 ncrt0.x30 -o sample <RET>
COPYRIGHT(C) XXXX(XXXX) RENESAS ELECTRONICS CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
```

<RET> : Means entering the return key.

Figure G.3 Example utl30 Command Line

G.2.2 Selecting Output Informations

To select outputs between "SBDATA declaration" and "SPECIAL page function declaration" in utl30, specify the options described below. If neither option is specified, an error is assumed for utl30.

(1) Output SBDATA declaration
- Option "-sb30"

(2) Output SPECIAL page Function declaration
- Option "-sp30"

Figure G.3 shows the sbutl command line options.

G.2.3 utl30 Command Line Options

The following information (input parameters) is needed in order to start utl30. Table G.1 shows the utl30 command line options.

Table G.1 utl30 Command Line Options

Option	Short form	Description
-all	None	[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.
-fsection	None	The variables and functions specified by #pragma SECTION are also included among those to be processed.
-fover_write	-fOW	Forcibly writes over the output file name specified with the -o option.
-o	None	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. No extensions can be specified. If the specified file already exists, the result is written to the standard output device.
-sb30	None	-sb30 -> Outputs SBDATA declaration. -sp30 -> Outputs SPECIAL page function declaration. To use utl30, always specify one of the two options. If neither option is specified, an error is assumed.
-sp=<number> --sp=<number>,<number>,... (two or more numbers) -sp=<number>-<number>	None	Does not use the specified number(s) as SPECIAL Page Function numbers. Use this option simultaneously with the -sb30 option.
-sp30	None	-sb30 -> Outputs SBDATA declaration. -sp30 -> Outputs SPECIAL page function declaration. To use utl30, always specify one of the two options. If neither option is specified, an error is assumed.
-Wstdout	None	Output the warning and error messages to the host machines standard output device.

-all

Makes all gobal variables vaild

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

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.

-sb30

Outputs SBDATA declaration

Function : Outputs SBDATA declaration. This option can be specified simultaneously with -sp30.

-sp30

Outputs SPECIAL page function declaration

Function : Outputs SPECIAL page function declaration. This option can be specified simultaneously with -sb30.

-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.

-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

Variables give below are excluded from SBDATA declaration.

- variables positioned in sections worked on by #pragma SECTION
- variables defined by #pragma ADDRESS
- variables defined by #pragma ROM
- const-qualified variables (except when -fconst_not_ROM or -fCNR has been specified for compilation)

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 ult30, and variables to be allocated are picked out of the remainder of the SB area.

G.5 Example of utl30 use

G.5.1 Generating a SBDATA declaration file

a. Generating a SBDATA declaration file

You can output a SBDATA declaration file by means of causing the SBDATA declaration utility utl30 to process files holding information as to the state of using variables.

Figure G.4 shows an example of making entries in utl30 , and Figure G.5 shows an example of SBDATA declaration file.

```
% utl30 -sb30 ncrt0.x30 -osbdata<RET>

%: Prompt
ncrt0.x30 : Name of absolute file
```

Figure G.4 Example utl30 Command Line

```
/*
* #pragma SBDATA Utility
*/
/* SBDATA Size [255] */
#pragma SBDATA      data3          /* size = (4) / ref = [ 2] */
#pragma SBDATA      data2          /* size = (1) / ref = [ 1] */
#pragma SBDATA      data1          /* size = (2) / ref = [ 1] */
                                        (1)               (2)
/*
* End of File
*/

(1)Size=() is size of data
(2)ref =() is access count of the variables
```

Figure G.5 SBDATA declaration File (sbdata.h)

You include the SBDATA declaration file generated above in a program as a header file .Figure G.6 shows an example of making setting in a SBDATA file.
Figure G.6 shows an example of making setting in a SBDATA file.

```
#include    "sbdata.h"

void          func(void)
{
              (ommit)
                  :
```

Figure G.6   Example of making settings in a SBDATA

b. Adjustment in an instance in which SB declaration is made in assembler

If the SB area is used as a result of the .sbsym declaration in an assembler routine, you need to adjust the file generated by utl30.

[assembler routine]

```
        .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] */
#pragma SBDATA    data2          /* size = (1) / ref = [ 1] */
          :
        (omitted)
          :
#pragma SBDATA    data1          /* size = (2) / ref = [ 1] */
/*
 * 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] */   /* Comments out*/
```

Figure G.7 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

It is possible to output SPECIAL page function declaration and SPECIAL page vector definition files by having the absolute module file (generated by using the option -finfo when compiling) processed by utl30, the SBDATA Declaration & SPECIAL Page Function Declaration Utility.
Figure G.8 shows an example of input for utl30. Figure G.9 shows an example of a SPECIAL page function declaration file. Figure G.10 shows an example of a SPECIAL page vector definition file.

```
% utl30   -sp30   ncrt0.x30   -o special<RET>

% : Prompt
ncrt0.x30 : Name of absolute file
```

Figure G.8 Example utl30 Command Line

```
/*
 * #pragma SPECIAL PAGE Utility
 */
/* SBDATA Size [255] */
#pragma   SPECIAL   255          func1    /* size = (100) / ref = [ 10] */
#pragma   SPECIAL   254          func2    /* size = (100) / ref = [ 7] */
#pragma   SPECIA    253          func3    /* size = (100) / ref = [ 5] */
                                           (1)                    (2)
/*
 * End of File
 */


(1) Indicates the function size.
(2) Indicates the reference frequency of function.
```

Figure G.9 SPECIAL Page Function declaration File (special.h)

You include the SPECIAL Page Function declaration file generated above in a program as a header file. Figure G.10 shows an example of making setting in a SPECIAL Page Function declaration File.

```
#include    "special.h"

void          func(void)
{
              (ommit)
                  :
```

Figure G.10   Example of making settings in a SPECIAL Page Function File

G.6 utl30 Error Messages

G.6.1 Error Messages

Table G.2 lists the utl30 calculation utility error messages and their countermeasures.

Table G.2 sbutl Error Messages

Error message	Contents of error and corrective action
ignore option '?'	• You specified an option that cannot be in used utl30. ⇒ Specify a proper option.
Illegal file extension'.XXX'	• Extension of input file is illegal. ⇒ Specify a proper file.
No input "x30" file specified	• No map file ⇒ Specify map file.
cannot open "x30" file 'file-name'	• Map file not found ⇒ Specify the correct input map file.
cannot close file 'file-name'	• input file cannot be closed ⇒ Specify the correct input file-name.
cannot open output file 'file-name'	• Output file cannot be close ⇒ Specify the correct output file-name.
not enough memory	• The extended memory is insufficient ⇒ Increase the extended memory
since 'file-name' file exist, it makes a standard output	• The 'file-name' specified with -o already exist. ⇒ Check the output file name. The file can be overwritten by specifying -fover_write simultaneously with the options.

G.6.2 Warning Messages

Table G.3 lists the sbutl utility warning messages and their countermeasures.

Table G.3 sbutl Warning Messages

Warning Message	Contents of warning and corrective action
confllict declare of 'variable	• The variable shown here is declared in multiple files with different storage classes, types, etc. ⇒ Check how this variable is declared.
confllict declare of 'function	• The function shown here is declared in multiple files with different storage classes, types, etc. ⇒ Check how this function is declared.

Appendix H Using gensni or the stack information File Creation Tool for Call Walker

Before Call Walker or the stack analysis tool of the High-performance Embedded Workshop can be used, you must have stack information files as the input files for it.
You use gensni or the stack information file creation tool for Call Walker to create these stack information files from the absolute module file.

H.1 Starting Call Walker

To start Call Walker, select "Call Walker" that is registered to the High-performance Embedded Workshop or select the tool from the Tools menu of the High-performance Embedded Workshop.
After starting Call Walker, choose Import Stack File from the File menu and select a stack information file as the input file for Call Walker.
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.

H.2 Outline of gensni

H.2.1 Processing Outline of gensni

gensni is the tool to create stack information files for Call Walker.
gensni generates a stack information file by processing the absolute module file (extension .x30). Before gensni can be used, there must be an absolute module file (extension .x30) available. Specify the compile option "-finfo" , "-g" during compilation to generate that file.
The processing flow of NC30 is shown in Figure H.1.

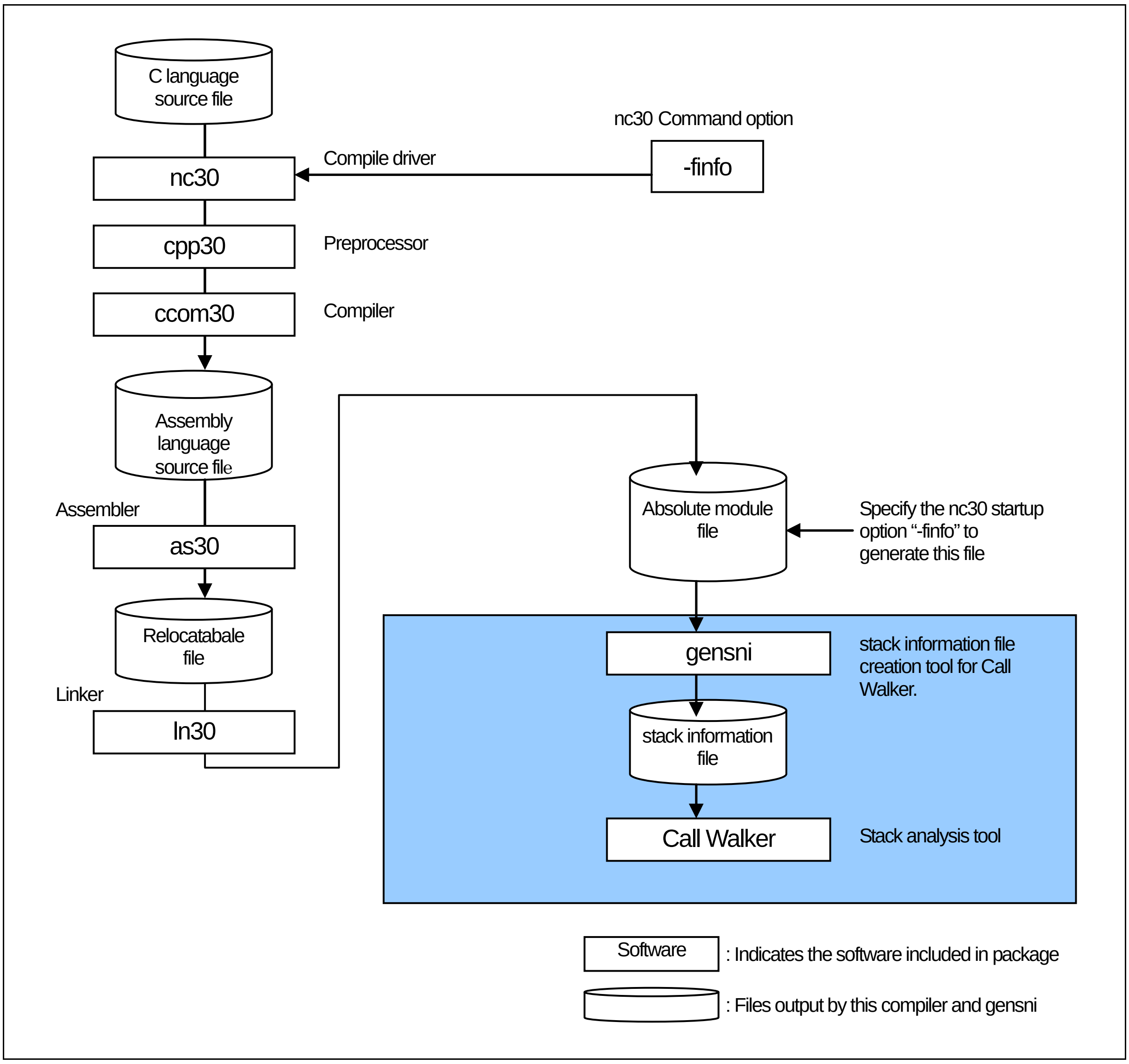


Figure H.1 Processing flow of NC30

H.3 Starting gensni

If Call Walker is started from the High-performance Embedded Workshop, gensni is automatically executed. However, if Call Walker is started from other than the High-performance Embedded Workshop, gensni is not automatically executed. In this case, start gensni from the Windows command prompt.

H.3.1 Input format

To start gensni, specify an input file name and startup option according to the input format shown below.

```
% gensni△[Command option]△Absolute module file(extension.x30)

% : Denotes the prompt
< > : Denotes the essential items.
[ ] : Denotes the items that need to be written when necessary.
△ : Denotes a space.
When writing multiple startup options, separate each with a space.
```

Figure H.2 gensni command input format

To use gensni, specify both of the following in the startup options of this compiler

- Inspector information output.. -finfo option

to generate absolute module files (extension ".x30").
An input example is shown below. In the input example here, the following option is specified in gensni.

- Information output to a specified file.. -o option

(By default, the information is output to a file named after the input file by changing the file extension from ".x30" to ".sni."

```
Generate an absolute module file :

% nc30 –finfo ncrt0.a30 sample.c <RET>
M16C Series, R8C Family Compiler V.X.XX Release XX
Copyright(C) XXXX(XXXX,XXXX,XXXX,XXXX). Renesas Electronics Corp.
and Renesas Solutions Corp., All rights reserved.

ncrt0.a30
sample.c

%

Generate stack information file:

%gensni -o sample ncrt0.x30<RET>

sample.sni is created.

%
```

Figure H.3 gensni command input example

H.3.2 Option References

The startup options of gensni are listed in Table H.1.

Table H.1 gensni Command option

Option	short form	function
-o file name	None	Specify a stack information file name. ● If this option is not specified, stack information file is named after the input file by changing its file extension to ".sni." ● If an extension is specified stack information file name, the specified extension is changed to ".sni." If no extensions are specified, the extension ".sni" is assumed.
-V	None	Shows the startup message of gensni and terminates processing without performing anything. No . stack information files are generated.

-o file

Specify a stack information file name

Function:

- If this option is not specified, stack information file is named after the input file by changing its file extension to ".sni."
- If no extensions are specified, the extension ".sni" is assumed.

Description: Use of this option permits you to change stack information file name as necessary. The extension can also be changed.

-V

Terminate processing after showing the startup message of gensni

Function: Shows the startup message of gensni and terminates processing without performing anything.

- No stack information files are generated.

M16C Series, R8C Family C Compiler Package V.5.45
C Compiler User's Manual

Publication Date: Nov. 1, 2010 Rev.3.00

Published by: Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi,
Kanagawa 211-8668 Japan

Edited by: Renesas Solutions Corp.

M16C Series, R8C Family
C Compiler Package V.5.45
C Compiler Use’s Manual

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan

RE10J1995-0300