NC30 V.5.20

C Compiler for R8C/Tiny, M16C/60,30,20,10 Series

User's Manual

Rev. 1.00
Sep 1, 2003
REJ10J0225-0100Z

Renesas Technology
www.renesas.com

- Microsoft, MS-DOS, Windows, and Windows NT are registered trademarks of Microsoft Corporation in the U.S. and other countries.
- Sun, Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. or other countries, and are used under license.
- Linux is a trademark of Linus Torvalds.
- Turbolinux and its logo are trademarks of Turbolinux, Inc.
- IBM and AT are registered trademarks of International Business Machines Corporation.
- Intel and Pentium are registered trademarks of Intel Corporation.
- Adobe, Acrobat, and Acrobat Reader are trademarks of Adobe Systems Incorporated.
- All other brand and product names are trademarks, registered trademarks or service marks of their respective holders.

Keep safety first in your circuit designs!

- Renesas Technology Corporation and Renesas Solutions Corporation put the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

- These materials are intended as a reference to assist our customers in the selection of the Renesas Technology product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation, Renesas Solutions Corporation or a third party.
- Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation and Renesas Solutions Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor for the latest product information before purchasing a product listed herein. The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors. Please also pay attention to information published by Renesas Technology Corporation and Renesas Solutions Corporation by various means, including the Renesas home page (http://www.renesas.com).
- When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability or other loss resulting from the information contained herein.
- Renesas Technology semiconductors 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 Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
- The prior written approval of Renesas Technology Corporation and Renesas Solutions Corporation is necessary to reprint or reproduce in whole or in part these materials.
- If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
- Please contact Renesas Technology Corporation or Renesas Solutions Corporation for further details on these materials or the products contained therein.

For inquiries about the contents of this document or product, fill in the text file the installer generates in the following directory and email to your local distributor.

¥SUPPORT¥Product-name¥SUPPORT.TXT

Renesas Tools Homepage http://www.renesas.com/en/tools

NC30 User's Manual

Contents

Contents

Preface

NC30 is the C compiler for the Renesas M16C/60,30,20,10 Series . NC30 converts programs written in C into assembly language source files for the M16C/60,30,20,10 Series. 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.

Terminology

The following terms are used in the NC30 User Manuals.

Term	Meaning
NC30	Compiler package included in M3T-NC30WA
nc30	Compile driver and its executable file
AS30	Assembler package included in M3T-NC30WA
as30	Relocatable macro assembler and its executable file
TM	Integrated development environment be attached to M3T-NC30WA
Professional version	Professional use compiler for full-scale programming
Entry version	Simplified compiler included in the starter kit, etc.

Description of Symbols

The following symbols are used in the NC30 manuals:

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

Additional descriptions are provided where other symbols are used.

NC30

User's Manual

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 eight executable files:

1.nc30 .... Compile driver
2.cpp30 .... Preprocessor
3.ccom30 .... Compiler
4.aopt30 .... Assembler Optimizer
5.StkViewer & stk .... STK viewer & stack size calculation Utility
(StkViewer is a GUI (Graphical User Interface) Utility.)
6.utl30 .... SBDATA declaration & SPECIAL page Function declaration Utility
7.MapViewer .... Map viewer (included for only the Windows(TM) version)
(MapViewer is a GUI (Graphical User Interface) Utility.)
(Items 4 to 7 are not included in the entry version.)

1.2 NC30 Processing Flow

Figure 1.1 illustrates the NC30 processing flow.

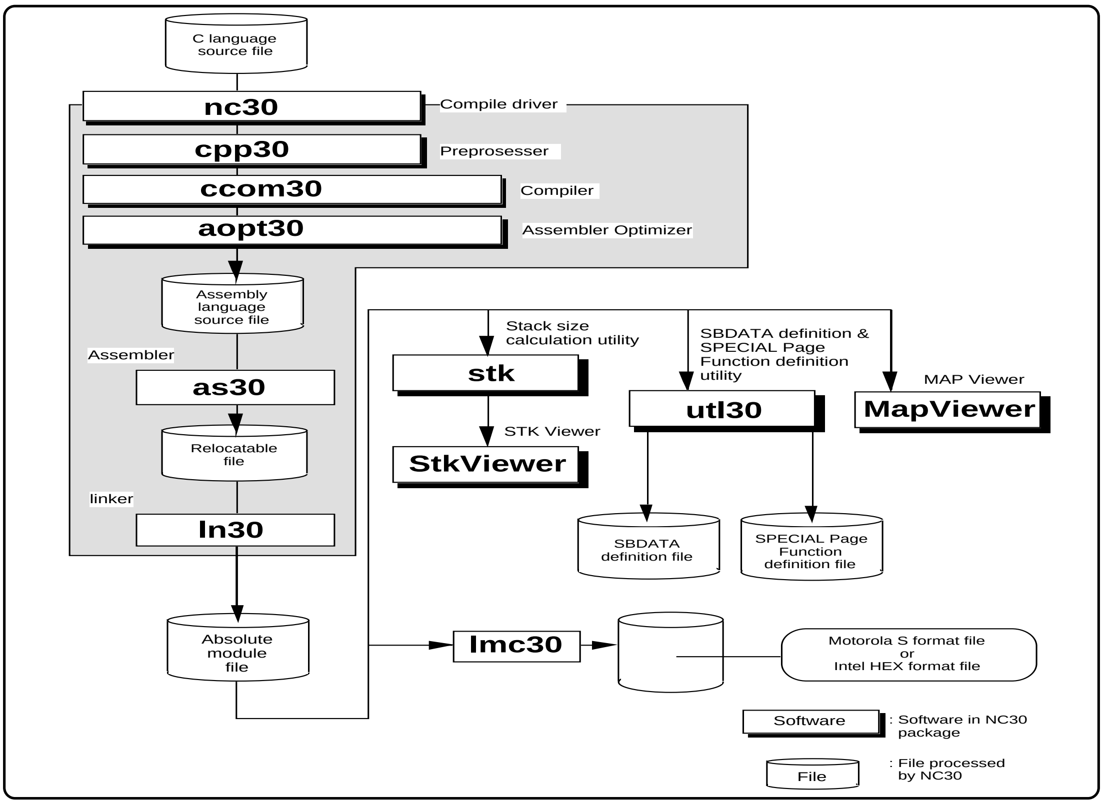


Figure1.1 NC30 Processing Flow

1. MapViewer is included for only the Windows(TM) version. To verify the map information while using the UNIX version, generate a map file with the linker and check that file for map information.

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.
(In the entry version, this option cannot be specified.)

1.2.5 StkViewer & stk

StkViewer is the execution file for the utility that graphically shows the stack size and the relationship of function calls needed for program operation. Also, stk is the execution file for the utility that analyzes the information required for StkViewer.

StkViewer calls stk to process the Inspector[*1] information added to the absolute module file (.x30), find the stack size and the relationship of function calls needed for program operation, and displays the result.

Also, by specifying information, if any, that could not be fully analyzed with only the Inspector information, StkViewer recalculates the stack size and the relationship of function calls and displays the result.

To use StkViewer & stk, specify the compile driver startup option -finfo when compiling, so that the Inspector information will be added to the absolute module file (.x30).

(In the entry version, this option cannot be specified.)

1.2.6 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.

1.2.7 MapViewer

MapViewer is the execution file for the map viewer. By processing the absolute module file (.x30), MapViewer graphically shows a post-link memory mapping.

To use MapViewer, specify the compile driver startup option -finfo when compiling, so that the absolute module file (.x30) will be generated.

Note that MapViewer is included for only the PC version. To verify the map information while using the UNIX version, generate a map file with the linker and check that file for map information.(In the entry version, this option cannot be specified.)

*1. The inspector information refers to one that is generated by NC30 when the compile option "-finfo" is specified.

1.3 Notes

Renesas Technology 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 Technology 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 NC308 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 case, write the instruction directly in the program using the ASM function.

In this compiler, the instructions which cannot be used may be generated for writing and read-out to the register of SFR area.

When accessing registers in the SFR area in C language, make sure that the same correct instructions are generated as done by using asm functions, regardless of the compiler's version and of whether optimizing options are used or not.

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.

[Example: C language description to SFR area]

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

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

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

1.4 Example Program Development

Figure 1.2 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 1.2.)

[1]The C source program AA.c is compiled using nc30, then assembled using as30 to create the relocatable object file AA.r30.

[2]The startup program ncrt0.a30 and the include file sect30.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.

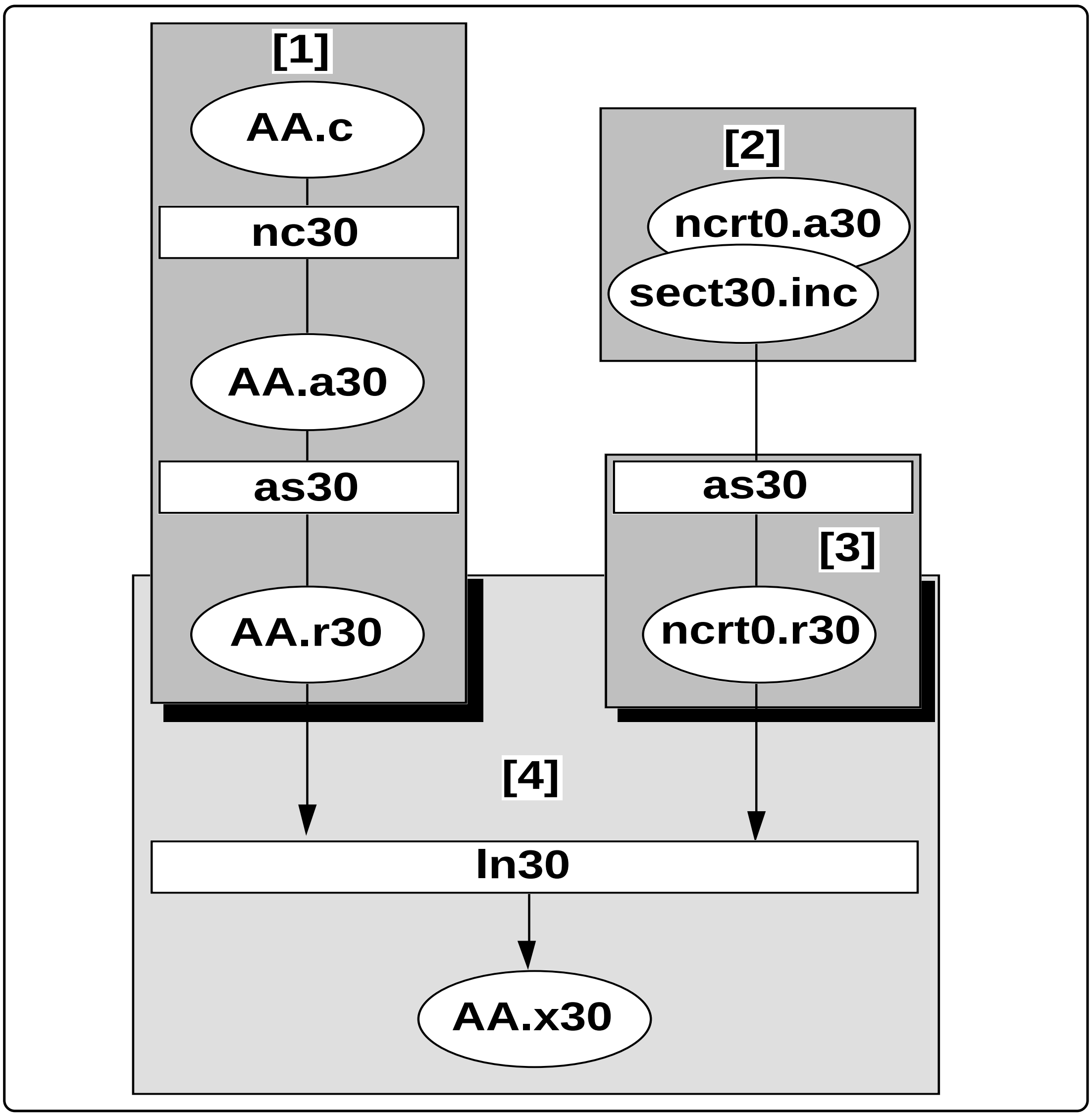


Figure 1.2 Program Development Flow

Figure 1.3 is an example make file containing the series of operations shown in Figure 1.2.

```
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 1.3 Example make File

Figure 1.4 shows the command line required for nc30 to perform the same operations as in the makefile shown in Figure 1.3.

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

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

*Specify ncrt0.a30 first ,when linking.

Figure 1.4 Example nc30 Command Line

1.5 NC30 Output Files

This chapter introduces the preprocess result C source program output when the sample program smp.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 1.5. Below, we show the contents of the files output when the C source file smp.c shown in Figure 1.6 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.

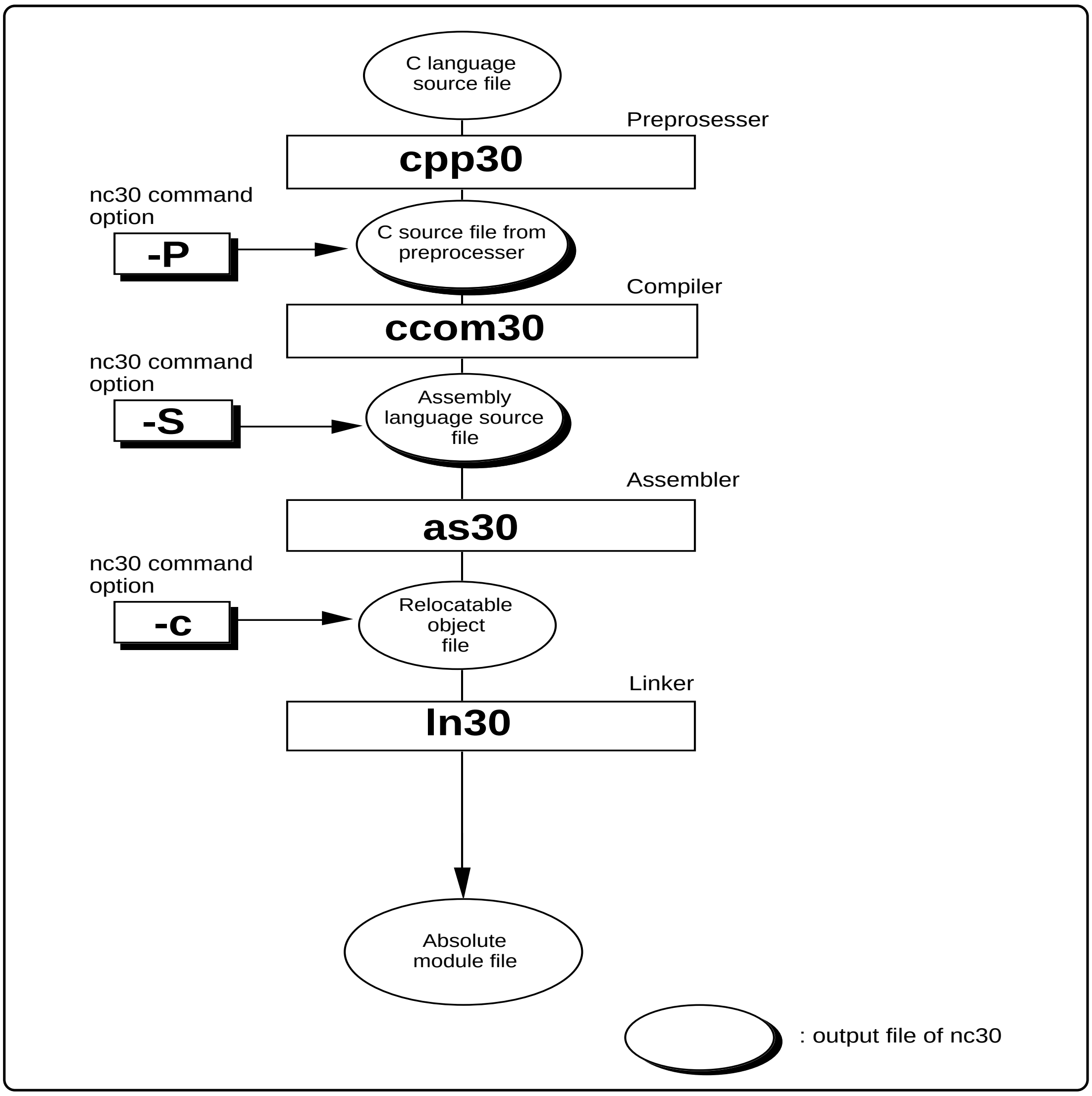


Figure 1.5 Relationship of nc30 Command Line Options and Output Files

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

void main()
{
          int flag;
          flag = CLR;

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

Figure 1.6 Example C Source File (smp.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.

Figures 1.7 and 1.8 are examples of file output.

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

typedef unsigned int size_t;

extern FILE _iob[];
```

Figure 1.7 Example Preprocessed C Source File (1) (smp.i)

```
int getc(FILE  *st);                                                    [1]
int getchar(void);
int putc(int c, FILE  *st);
int putchar(int c);
int feof(FILE  *st);
int ferror(FILE  *st);
int fgetc(FILE  *st);
char * fgets(char  *s, int n, FILE  *st);
int fputc(int c, FILE  *st);
int fputs(const char  *s, FILE  *st);
size_t fread(void  *ptr, size_t size, size_t nelem, FILE  *st);
          :
      (omitted)
          :
int ungetc(int c, FILE  *st);
int printf(const char  *format, ...);
int fprintf(FILE  *st, const char  *format, ...);
int sprintf(char  *s, const char  *format, ...);
          :
      (omitted)
          :
extern  int     init_dev(FILE  *, 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  char     *_print( int(*)(), char  *, int **, int * );
```

```
void main()                                 [2]
{
        int flag;
        flag = 0 ;   ⇐[3]


         printf("flag = %d\n",flag);  ⇐[4]

}
```

Figure 1.8 Example Preprocessed C Source File (2) (smp.i)

Let's look at the contents of the preprocessed C source file.
Items [1] to [4] correspond to [1] to [4] in Figures 1.7 and 1.8.

- [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

Figures 1.9 and 1.10 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'

;## M16C/60 C Compiler   OUTPUT
;## ccom30 Version X.XX.XX
;## COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
;## ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;## Compile Start Time Thu April 10 18:40:11 1995,1996,1997,1998,1999,2000,2001,2002,2003

;## COMMAND_LINE: ccom30  smp.i -o ./smp.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

        .section        program
        ._file  'smp.c'
        ._line  6
;## # C_SRC :   {
        .glb    _main
_main:
        enter   #02H
        ._line  8
;## # C_SRC :            flag = CLR;
```

Figure 1.9 Example Assembly Language Source File "smp.a30" (1/2)

```
        mov.w   #0000H,-2[FB]   ;  flag
        ._line  11
;## # C_SRC :             printf("flag = %d\n",flag);  ⇐[2]
        push.w  -2[FB]  ;  flag
        push.w  #___T0>>16
        push.w  #(___T0&0FFFFH)
        jsr     _printf
        add.b   #06H,SP
        ._line  13
;## # C_SRC :   }
        exitd

          :
      (omitted)
          :

        .glb    _sscanf
        .glb    _scanf
        .glb    _fscanf
        .glb    _sprintf
        .glb    _fprintf
        .glb    _printf

          :
      (omitted)
          :

        .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     ;  '%'

          :
      (omitted)
          :

        .END

;## Compile End Time Thu May  5 18:40:11 2000
```

Figure 1.10 Example Assembly Language Source File "smp.a30" (2/2)

Let's look at the contents of the assembly language source files. Items [1] to [2] correspond to [1] to [2] in Figure 1.9 and 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 comments

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.a30 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 .... -o
*Specifies output of list file (extension .lst) at assembling .... -as30 "-l"
*Specifies output of map file (extension .map) at linking .... -ln30 "-ms"

% nc30Δ[*command-line-option*]Δ[*assembly-language-source-file-name*]Δ
[*relocatable-object-file-name*]Δ<*C-source-file-name*>

% : Prompt
< > : Mandatory item
[ ] : Optional item
Δ : 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 Windows (TM), etc.

a. Command file input format

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

% : 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>
sample<CR>

<CR>: Denotes carriage return.

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

There are the following priorities in the option about control of a 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 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 relocatable 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.

Table 2.1 Options for Controlling Compile Driver

Option	Function
-c	Creates a relocatable file (extension .r30) and ends processing [*1]
-D*identifier*	Defines an identifier. Same function as #define.
-I*directory*	Specifies the directory containing the file(s) specified in #include. You can specify up to 8 directories.
-E	Invokes only preprocess commands and outputs result to standard output.[*1]
-P	Invokes only preprocess commands and creates a file (extension .i). [*1]
-S	Creates an assembly language source file (extension .a30) and ends processing.[*1]
-U *predefined macro*	Undefines the specified predefined macro.
-silent	Suppresses the copyright message display at startup.
-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).

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
-o*filename*	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.
-dir	Specifies the destination directory of the file(s) (absolute module file, map file, etc.) generated by ln30.

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)

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.

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. In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-gno_reg	Suppresses the output of debugging information for register variables. In the entry version, this option cannot be specified.
-gold	outputs debugging information for old version debuggers and third-party debuggers In the entry version, this option cannot be 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.	Maximum optimization of speed and ROM size
-OR	None.	Maximum optimization of ROM size followed by speed
-OS	None.	Maximum optimization of speed followed by ROM size
-Oconst	-OC	Performs optimization by replacing references to the const-qualified external variables with constants
-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_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.
-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_logical _or_combine	-ONLOC	Suppresses the optimization that puts consecutive ORs together.
-Ono_asmpot	-ONA	Inhibits starting the assembler optimizer "aopt30."
-Ostatic_to_inline	-OSTI	A static function is treated as an inline function.

f. Generated Code Modification Options

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

Table 2.6 (1/2) Generated Code Modification Options

Option	Short form	Description
-fansi	None.	Makes -fnot_reserve_far_and_near, -fnot_reserve_asm, and -fextend_to_int valid. In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fnot_reserve_asm	-fNRA	Exclude asm from reserved words. (Only _asm is valid.) In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fnot_reserve_far_and_near	-fNRFAN	Exclude far and near from reserved words. (Only _far and _near are valid.) In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fnot_reserve_inline	-fNRI	Exclude far and near from reserved words. (Only _inline is made a reserved word.) In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fextend_to_int	-fETI	Performs operation after extending char-type data to the int type. (Extended according to ANSI standards.)*1 In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fchar_enumerator	-fCE	Handles the enumerator type as an unsigned char type, not as an int type.
-fno_even	-fNE	Allocate all data to the odd section , with no separating odd data from even data when outputting .
-ffar_RAM	-fFRAM	Changes the default attribute of RAM data to far.
-fnear_ROM	-fNROM	Changes the default attribute of ROM data to near. In the entry version, this option cannot be specified.
-fconst_not_ROM	-fCNR	Does not handle the types specified by const as ROM data.

*1. char-type data or signed char-type data evaluated under ANSI rules is always extended to int-type 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 2.6 (2/2) Generated Code Modification Options

Option	Short form	Description
-fnot_address_volatile	-fNAV	Does not regard the variables specified by #pragma ADDRESS (#pragma EQU) as those specified by volatile.
-fsmall_array	-fSA	When referencing a far-type array, this option calculates subscripts in 16 bits if the total size of the array is within 64K bytes. In the entry version, this option cannot be specified.
-fenable_register	-fER	Make register storage class available
-fno_align	-fNA	Does not align the start address of the function. In the entry version, this option cannot be specified.
-fJSRW	None.	Changes the default instruction for calling functions to JSR.W.
-fbit	-fB	Outputs a 1-bit manipulate instruction to all variables arranged in 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 byte.
-fuse_DIV	-fUD	This option changes generated code for divide operation. In the entry version, this option cannot be specified.
-finfo	None	Outputs the information required for the Inspector, STK Viewer, Map Viewer, and utl30 to the absolute module file (.x30). In the entry version, this option cannot be specified.
-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.

g. Library Specifying Option

Table 2.7 lists the startup options you can use to specify a library file.

Table 2.7 Library Specifying Option

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

h. Warning Options

Table 2.8 shows the command line options for outputting warning messages for contraventions of nc30 language specifications.

Table 2.8 Warning Options

Option	Short form	Function
-Wnon_prototype	-WNP	Outputs warning messages for functions without prototype declarations.
-Wunknown_pragma	-WUP	Outputs warning messages for non-supported #pragma.
-Wno_stop	-WNS	Prevents the compiler stopping when an error occurs.
-Wstdout	None.	Outputs error messages to the host machine's standard output (stdout).
-Werror_file<file name>	-WEF	Outputs error messages to the specified file.
-Wstop_at_warning	-WSAW	Stops the compiling process when a warning occurs.
-Wnesting_comment	-WNC	Outputs a warning for a comment including */ .
-Wccom_max_warnings = Warning Count	-WCMW	This option allows you to specify an upper limit for the number of warnings output by ccom30.
-Wall	None.	Displays message for all detectable warnings(however, not including alarms output by -Wlarge_to_small and -Wno_used_argument).
-Wmake_tagfile	-WMT	Outputs error messages to the tag file of source-file by source-file.
-Wuninitialize_variable	-WUV	Outputs a warning about auto variables that have not been initialized.
-Wlarge_to_small	-WLTS	Outputs a warning about the tacit transfer of variables in descending sequence of size.
-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.
-Wno_used_argument	-WNUA	Outputs a warning for unused argument of functions.

i. Assemble and Link Options

Table 2.9 shows the command line options for specifying as30 and ln30 options.

Table 2.9 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. In the entry version, this option cannot be specified.
-ln30Δ<*option*>	Specifies options for the ln30 assemble command. If you specify two or more options, enclose them in double quotes. In the entry version, this option cannot be specified.

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:

1. ncrt0.a30
   Write a program which is executed immediately after reset.T

2. sect30.inc
   Included from ncrt0.a30, this file defines section locations (memory mapping).

Figures 2.6 to 2.9 show the ncrt0.a30 source program list. Figures 2.10 to 2.13 show the sect30.inc source program list.

```
;***************************************************************************
;
; C COMPILER for M16C/60,20
;* COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
;* ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
; ncrt0.a30 : NC30 startup program
;
; This program is applicable when using the basic I/O library
;
; $Id: ncrt0.a30,v 1.16 2003/03/27 10:57:43 XXXXXXXX Exp $
;
;***************************************************************************

;         .glb  __BankSelect
;__BankSelect  .equ  0BH
;---------------------------------------------------------------------
; HEEP SIZE definition
;---------------------------------------------------------------------
HEAPSIZE  .equ  300h                              ⇐[1]

;---------------------------------------------------------------------
; STACK SIZE definition
;---------------------------------------------------------------------
STACKSIZE .equ  300h                              ⇐[2]

;---------------------------------------------------------------------
; INTERRUPT STACK SIZE definition
;---------------------------------------------------------------------
ISTACKSIZE     .equ  300h                         ⇐[3]

;---------------------------------------------------------------------
; INTERRUPT VECTOR ADDRESS  definition
;---------------------------------------------------------------------
VECTOR_ADR     .equ  0ffd00h                      ⇐[4]
```

[1]defines the heap size.
[2]defines the user stack size.
[3]defines the interrupt stack size.
[4]defines the start address of interrupt vector table.

Figure 2.6 Startup Program List (1)(ncrt0.a30 1/4)

```
;---------------------------------------------------------------
; special page definition
;---------------------------------------------------------------
;       macro define for special page
;
;Format:
;   SPECIAL    number
;

SPECIAL  .macro     NUM
    .org  0FFFFEH-(NUM*2)
    .glb  __SPECIAL_@NUM
    .word __SPECIAL_@NUM  & 0FFFFH
.endm
;---------------------------------------------------------------------
; Section allocation
;---------------------------------------------------------------------
    .list OFF
    .include sect30.inc                                  ⇐[5]
    .list ON

;---------------------------------------------------------------------
; SBDATA area definition
;---------------------------------------------------------------------
    .glb  __SB__
__SB__    .equ  data_SE_top

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

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

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


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

[5]Includes sect30.inc

Figure 2.7   Startup Program List (2) (ncrt0.a30 2/4)

```
;====================================================================
; Interrupt section start
;---------------------------------------------------------------------
    .insf start,S,0
    .glb  start
    .section   interrupt
start:                                                ⇐[6]
;---------------------------------------------------------------------
; after reset,this program will start
;---------------------------------------------------------------------
    ldc   #istack_top,     isp   ;set istack pointer
    mov.b #02h,0ah
;   bset  1,0ah
    mov.b #00h,04h         ;set processer mode   ⇐[7]
;   bclr  1,0ah
    mov.b #00h,0ah
    ldc   #0080h,    flg                         ⇐[8]
    ldc   #stack_top,sp    ;set stack pointer
    ldc   #data_SE_top,    sb    ;set sb register
    ldintb      #VECTOR_ADR

;====================================================================
; NEAR area initialize.
;--------------------------------------------------------------------
; bss zero clear
;--------------------------------------------------------------------
    N_BZERO     bss_SE_top,bss_SE                ⇐[9]
    N_BZERO     bss_SO_top,bss_SO
    N_BZERO     bss_NE_top,bss_NE
    N_BZERO     bss_NO_top,bss_NO

;---------------------------------------------------------------------
; initialize data section
;---------------------------------------------------------------------
    N_BCOPY data_SEI_top,data_SE_top,data_SE     ⇐[10]
    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

;====================================================================
; FAR area initialize.
;---------------------------------------------------------------------
; bss zero clear
;---------------------------------------------------------------------
    BZERO bss_FE_top,bss_FE                      ⇐[11]
    BZERO bss_FO_top,bss_FO

;---------------------------------------------------------------------
; Copy edata_E(O) section from edata_EI(OI) section
;---------------------------------------------------------------------
    BCOPY data_FEI_top,data_FE_top,data_FE       ⇐[12]
    BCOPY data_FOI_top,data_FO_top,data_FO

    ldc   #stack_top,sp
    .stk  -40
```

[6]After a reset, execution starts from this label (start)

[7]Sets processor operating mode

[8]Sets IPL and each flags.

[9]Clears the near and SBDATA bss section (to zeros)

[10]Moves the initial values of the near and SBDATA data section to RAM

[11]Clears the far bss section (to zeros) *1

[12]Moves the initial values of the far data section to RAM *1

Figure 2.8 Startup Program List (3) (ncrt0.a30 3/4)

*1. Comment out this line if no far area is used.

```
;====================================================================
; heap area initialize
;---------------------------------------------------------------------
    .glb  __mbase                                        ⇐[13]
    .glb  __mnext
    .glb  __msize
    mov.w #(heap_top&0FFFFH), __mbase
    mov.w #(heap_top>>16), __mbase+2
    mov.w #(heap_top&0FFFFH), __mnext
    mov.w #(heap_top>>16), __mnext+2
    mov.w #(HEAPSIZE&0FFFFH), __msize
    mov.w #(HEAPSIZE>>16), __msize+2

;====================================================================
; Initialize standard I/O
;---------------------------------------------------------------------
    .glb  _init                                          ⇐[14]
    .call _init,G
    jsr.a _init

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

    .glb  _main                                          ⇐[15]
    jsr.a _main

;====================================================================
; exit() function
;---------------------------------------------------------------------
    .glb  _exit                                          ⇐[16]
    .glb  $exit
_exit:                 ; End program
$exit:
    jmp   _exit
    .einsf
;====================================================================
; dummy interrupt function
;---------------------------------------------------------------------
dummy_int:                                               ⇐[17]
    reit
    .end
;***************************************************************************
;
; C COMPILER for M16C/60,20
;* COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
;* ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
;***************************************************************************
```

[13]Initializes the heap area. Comment out this line if no memory management function is used.

[14]Calls the init function, which initializes standard I/O. Comment out this line if no I/O function is used.

[15]Calls the 'main' function. *1

[16]exit function

[17]Dummy interrupt processing function

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

*1. Interrupt is not enable, when calls 'main' function. Therefore, permits interrupt by FSET command, when uses interrupt function.

```
;****************************************************************************
;
; C Compiler for M16C/60,20
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
; Written by X.XXXXXXXX
;
; sect30.inc     : section definition
; This program is applicable when using the basic I/O library
;
; $Id: sect30.inc,v 1.14 2003/03/27 10:57:43 XXXXXXXX Exp $
;
;****************************************************************************
;---------------------------------------------------------------
;
;   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   stack,DATA
    .blkb STACKSIZE
stack_top:

    .blkb ISTACKSIZE
istack_top:


;---------------------------------------------------------------
;   heap section
;---------------------------------------------------------------
    .section   heap,DATA
heap_top:
    .blkb HEAPSIZE
```

Figure 2.10 Startup Program List (5) (sect30.inc 1/8)

```
;---------------------------------------------------------------
; Near ROM data area
;---------------------------------------------------------------
    .section    rom_NE,ROMDATA,ALIGN
rom_NE_top:

    .section    rom_NO,ROMDATA
rom_NO_top:

;---------------------------------------------------------------
; Far RAM data area
;---------------------------------------------------------------
    .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:


;---------------------------------------------------------------
; Far ROM data area
;---------------------------------------------------------------
    .section    rom_FE,ROMDATA
    .org        0F0000H
rom_FE_top:

    .section    rom_FO,ROMDATA
rom_FO_top:

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

    .section    data_SOI,ROMDATA
data_SOI_top:

    .section    data_NEI,ROMDATA
data_NEI_top:

    .section    data_NOI,ROMDATA
data_NOI_top:

    .section    data_FEI,ROMDATA
data_FEI_top:

    .section    data_FOI,ROMDATA
data_FOI_top:

;---------------------------------------------------------------
; Switch Table Section
;---------------------------------------------------------------
    .section         switch_table,ROMDATA
switch_table_top:


;---------------------------------------------------------------
; code area
;---------------------------------------------------------------

    .section    program

    .section    interrupt
    ;.org ;must be set internal ROM area
```

Figure 2.11 Startup Program List (6) (sect30.inc 2/8)

```
    .section    program_S

;---------------------------------------------------------------
; variable vector section
;---------------------------------------------------------------
    .section    vector          ; variable vector table
    .org  VECTOR_ADR

.if M62TYPE==1
    .lword      dummy_int        ; BRK (vector 0)
    .lword      dummy_int        ;     (vector 1)
    .lword      dummy_int        ;     (vector 2)
    .lword      dummy_int        ;     (vector 3)
    .lword      dummy_int        ; int3(for user)(vector 4)
    .lword      dummy_int        ; timerB5(for user)(vector 5)
    .lword      dummy_int        ; timerB4(for user)(vector 6)
    .lword      dummy_int        ; timerB3(for user)(vector 7)
    .lword      dummy_int        ; si/o4 /int5(for user)(vector 8)
    .lword      dummy_int        ; si/o3 /int4(for user)(vector 9)
    .lword      dummy_int        ; Bus collision detection(for user)(v10)
    .lword      dummy_int        ; DMA0(for user)(vector 11)
    .lword      dummy_int        ; DMA1(for user)(vector 12)
    .lword      dummy_int        ; Key input interrupt(for user)(vect 13)
    .lword      dummy_int        ; A-D(for user)(vector 14)
    .lword      dummy_int        ; uart2 transmit(for user)(vector 15)
    .lword      dummy_int        ; uart2 receive(for user)(vector 16)
    .lword      dummy_int        ; uart0 transmit(for user)(vector 17)
    .lword      dummy_int        ; uart0 receive(for user)(vector 18)
    .lword      dummy_int        ; uart1 transmit(for user)(vector 19)
    .lword      dummy_int        ; uart1 receive(for user)(vector 20)
    .lword      dummy_int        ; timer A0(for user)(vector 21)
    .lword      dummy_int        ; timer A1(for user)(vector 22)
    .lword      dummy_int        ; timer A2(for user)(vector 23)
    .lword      dummy_int        ; timer A3(for user)(vector 24)
    .lword      dummy_int        ; timer A4(for user)(vector 25)
    .lword      dummy_int        ; timer B0(for user)(vector 26)
    .lword      dummy_int        ; timer B1(for user)(vector 27)
    .lword      dummy_int        ; timer B2(for user)(vector 28)
    .lword      dummy_int        ; int0 (for user)(vector 29)
    .lword      dummy_int        ; int1 (for user)(vector 30)
    .lword      dummy_int        ; int2 (for user)(vector 31)
.else
    .lword      dummy_int        ; vector 0 (BRK)
    .lword      dummy_int        ; vector 1
    .lword      dummy_int        ; vector 2
    .lword      dummy_int        ; vector 3
    .lword      dummy_int        ; vector 4
    .lword      dummy_int        ; vector 5
    .lword      dummy_int        ; vector 6
    .lword      dummy_int        ; vector 7
    .lword      dummy_int        ; vector 8
    .lword      dummy_int        ; vector 9
    .lword      dummy_int        ; vector 10
    .lword      dummy_int        ; DMA0 (for user) (vector 11)
    .lword      dummy_int        ; DMA1 2 (for user) (vector 12)
    .lword      dummy_int        ; input key (for user) (vector 13)
    .lword      dummy_int        ; AD Convert (for user) (vector 14)
    .lword      dummy_int        ; vector 15
    .lword      dummy_int        ; vector 16
    .lword      dummy_int        ; uart0 trance (for user) (vector 17)
    .lword      dummy_int        ; uart0 receive (for user) (vector 18)
    .lword      dummy_int        ; uart1 trance (for user) (vector 19)
    .lword      dummy_int        ; uart1 receive (for user) (vector 20)
    .lword      dummy_int        ; TIMER A0 (for user) (vector 21)
    .lword      dummy_int        ; TIMER A1 (for user) (vector 22)
    .lword      dummy_int        ; TIMER A2 (for user) (vector 23)
    .lword      dummy_int        ; TIMER A3 (for user) (vector 24)
    .lword   dummy_int           ; TIMER A4 (for user) (vector 25)
    .lword      dummy_int        ; TIMER B0 (for user) (vector 26)
    .lword      dummy_int        ; TIMER B1 (for user) (vector 27)
    .lword      dummy_int        ; TIMER B2 (for user) (vector 28)
    .lword      dummy_int        ; INT0 (for user) (vector 29)
    .lword      dummy_int        ; INT1 (for user) (vector 30)
    .lword      dummy_int        ; INT2 (for user) (vector 31)
```

Figure 2.12 Startup Program List (7) (sect30.inc 3/8)

```
.endif
    .lword     dummy_int        ; vector 32 (for user or MR30)
    .lword     dummy_int        ; vector 33 (for user or MR30)
    .lword     dummy_int        ; vector 34 (for user or MR30)
    .lword     dummy_int        ; vector 35 (for user or MR30)
    .lword     dummy_int        ; vector 36 (for user or MR30)
    .lword     dummy_int        ; vector 37 (for user or MR30)
    .lword     dummy_int        ; vector 38 (for user or MR30)
    .lword     dummy_int        ; vector 39 (for user or MR30)
    .lword     dummy_int        ; vector 40 (for user or MR30)
    .lword     dummy_int        ; vector 41 (for user or MR30)
    .lword     dummy_int        ; vector 42 (for user or MR30)
    .lword     dummy_int        ; vector 43 (for user or MR30)
    .lword  dummy_int           ; vector 44 (for user or MR30)
    .lword     dummy_int        ; vector 45 (for user or MR30)
    .lword     dummy_int        ; vector 46 (for user or MR30)
    .lword     dummy_int        ; vector 47 (for user or MR30)
    .lword     dummy_int        ; vector 48
    .lword     dummy_int        ; vector 49
    .lword     dummy_int        ; vector 50
    .lword     dummy_int        ; vector 51
    .lword     dummy_int        ; vector 52
    .lword     dummy_int        ; vector 53
    .lword     dummy_int        ; vector 54
    .lword     dummy_int        ; vector 55
    .lword     dummy_int        ; vector 56
    .lword     dummy_int        ; vector 57
    .lword     dummy_int        ; vector 58
    .lword     dummy_int        ; vector 59
    .lword     dummy_int        ; vector 60
    .lword     dummy_int        ; vector 61
    .lword     dummy_int        ; vector 62
    .lword     dummy_int        ; vector 63

;================================================================
; fixed vector section
;----------------------------------------------------------------
    .section   fvector                ; fixed vector table
;================================================================
; special page defination
;----------------------------------------------------------------
;   macro is defined in ncrt0.a30
;   Format: SPECIAL number
;
;----------------------------------------------------------------
;   SPECIAL 255
;   SPECIAL 254
;   SPECIAL 253
;   SPECIAL 252
;   SPECIAL 251
;   SPECIAL 250
;   SPECIAL 249
;   SPECIAL 248
;   SPECIAL 247
;   SPECIAL 246
;   SPECIAL 245
;   SPECIAL 244
;   SPECIAL 243
;   SPECIAL 242
;   SPECIAL 241
;   SPECIAL 240
;   SPECIAL 239
;   SPECIAL 238
;   SPECIAL 237
;   SPECIAL 236
;   SPECIAL 235
;   SPECIAL 234
;   SPECIAL 233
;   SPECIAL 232
;   SPECIAL 231
;   SPECIAL 230
;   SPECIAL 229
;   SPECIAL 228
;   SPECIAL 227
;   SPECIAL 226
```

Figure 2.13   Startup Program List (8) (sect30.inc 4/8)

```
;   SPECIAL 225
;   SPECIAL 224
;   SPECIAL 223
;   SPECIAL 222
;   SPECIAL 221
;   SPECIAL 220
;   SPECIAL 219
;   SPECIAL 218
;   SPECIAL 217
;   SPECIAL 216
;   SPECIAL 215
;   SPECIAL 214
;   SPECIAL 213
;   SPECIAL 212
;   SPECIAL 211
;   SPECIAL 210
;   SPECIAL 209
;   SPECIAL 208
;   SPECIAL 207
;   SPECIAL 206
;   SPECIAL 205
;   SPECIAL 204
;   SPECIAL 203
;   SPECIAL 202
;   SPECIAL 201
;   SPECIAL 200
;   SPECIAL 199
;   SPECIAL 198
;   SPECIAL 197
;   SPECIAL 196
;   SPECIAL 195
;   SPECIAL 194
;   SPECIAL 193
;   SPECIAL 192
;   SPECIAL 191
;   SPECIAL 190
;   SPECIAL 189
;   SPECIAL 188
;   SPECIAL 187
;   SPECIAL 186
;   SPECIAL 185
;   SPECIAL 184
;   SPECIAL 183
;   SPECIAL 182
;   SPECIAL 181
;   SPECIAL 180
;   SPECIAL 179
;   SPECIAL 178
;   SPECIAL 177
;   SPECIAL 176
;   SPECIAL 175
;   SPECIAL 174
;   SPECIAL 173
;   SPECIAL 172
;   SPECIAL 171
;   SPECIAL 170
;   SPECIAL 169
;   SPECIAL 168
;   SPECIAL 167
;   SPECIAL 166
;   SPECIAL 165
;   SPECIAL 164
;   SPECIAL 163
;   SPECIAL 162
;   SPECIAL 161
;   SPECIAL 160
;   SPECIAL 159
;   SPECIAL 158
;   SPECIAL 157
;   SPECIAL 156
;   SPECIAL 155
;   SPECIAL 154
;   SPECIAL 153
;   SPECIAL 152
;   SPECIAL 151
```

Figure 2.14 Startup Program List (9) (sect30.inc 5/8)

```
;   SPECIAL 150
;   SPECIAL 149
;   SPECIAL 148
;   SPECIAL 147
;   SPECIAL 146
;   SPECIAL 145
;   SPECIAL 144
;   SPECIAL 143
;   SPECIAL 142
;   SPECIAL 141
;   SPECIAL 140
;   SPECIAL 139
;   SPECIAL 138
;   SPECIAL 137
;   SPECIAL 136
;   SPECIAL 135
;   SPECIAL 134
;   SPECIAL 133
;   SPECIAL 132
;   SPECIAL 131
;   SPECIAL 130
;   SPECIAL 129
;   SPECIAL 128
;   SPECIAL 127
;   SPECIAL 126
;   SPECIAL 125
;   SPECIAL 124
;   SPECIAL 123
;   SPECIAL 122
;   SPECIAL 121
;   SPECIAL 120
;   SPECIAL 119
;   SPECIAL 118
;   SPECIAL 117
;   SPECIAL 116
;   SPECIAL 115
;   SPECIAL 114
;   SPECIAL 113
;   SPECIAL 112
;   SPECIAL 111
;   SPECIAL 110
;   SPECIAL 109
;   SPECIAL 108
;   SPECIAL 107
;   SPECIAL 106
;   SPECIAL 105
;   SPECIAL 104
;   SPECIAL 103
;   SPECIAL 102
;   SPECIAL 101
;   SPECIAL 100
;   SPECIAL 99
;   SPECIAL 98
;   SPECIAL 97
;   SPECIAL 96
;   SPECIAL 95
;   SPECIAL 94
;   SPECIAL 93
;   SPECIAL 92
;   SPECIAL 91
;   SPECIAL 90
;   SPECIAL 89
;   SPECIAL 88
;   SPECIAL 87
;   SPECIAL 86
;   SPECIAL 85
;   SPECIAL 84
;   SPECIAL 83
;   SPECIAL 82
;   SPECIAL 81
;   SPECIAL 80
;   SPECIAL 79
;   SPECIAL 78
;   SPECIAL 77
;   SPECIAL 76
```

Figure 2.15 Startup Program List (10) (sect30.inc 6/8)

```
;   SPECIAL 75
;   SPECIAL 74
;   SPECIAL 73
;   SPECIAL 72
;   SPECIAL 71
;   SPECIAL 70
;   SPECIAL 69
;   SPECIAL 68
;   SPECIAL 67
;   SPECIAL 66
;   SPECIAL 65
;   SPECIAL 64
;   SPECIAL 63
;   SPECIAL 62
;   SPECIAL 61
;   SPECIAL 60
;   SPECIAL 59
;   SPECIAL 58
;   SPECIAL 57
;   SPECIAL 56
;   SPECIAL 55
;   SPECIAL 54
;   SPECIAL 53
;   SPECIAL 52
;   SPECIAL 51
;   SPECIAL 50
;   SPECIAL 49
;   SPECIAL 48
;   SPECIAL 47
;   SPECIAL 46
;   SPECIAL 45
;   SPECIAL 44
;   SPECIAL 43
;   SPECIAL 42
;   SPECIAL 41
;   SPECIAL 40
;   SPECIAL 39
;   SPECIAL 38
;   SPECIAL 37
;   SPECIAL 36
;   SPECIAL 35
;   SPECIAL 34
;   SPECIAL 33
;   SPECIAL 32
;   SPECIAL 31
;   SPECIAL 30
;   SPECIAL 29
;   SPECIAL 28
;   SPECIAL 27
;   SPECIAL 26
;   SPECIAL 25
;   SPECIAL 24
;   SPECIAL 23
;   SPECIAL 22
;   SPECIAL 21
;   SPECIAL 20
;   SPECIAL 19
;   SPECIAL 18
```

Figure 2.16 Startup Program List (11) (sect30.inc 7/8)

```
;
;===============================================================
; fixed vector section
;---------------------------------------------------------------
    .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
;
;*******************************************************************************
;
;   C Compiler for M16C/60,20
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
;*******************************************************************************
```

Figure 2.17 Startup Program List (12) (sect30.inc 8/8)

2.2.2 Customizing the Startup Program

a. Overview of Startup Program Processing

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 and 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 2.18 summarizes the steps required to modify the startup programs to match the target system.

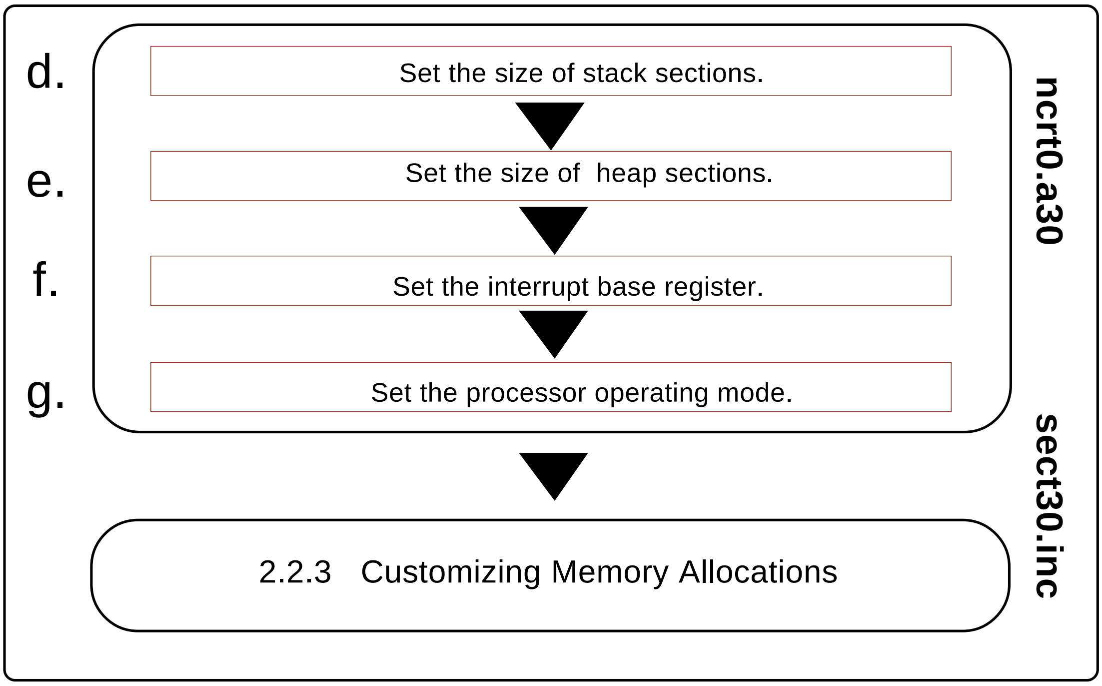


Figure 2.14 Example Sequence for Modifying Startup Programs

c. Examples of startup modifications that require caution

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

The init function*1 initializes the M16C/60 Series I/O. It is called before main in ncrt0.a30. Figure 2.19 shows the part where the init function is called.

If your application program does not use standard I/O, comment out the init function call from ncrt0.a30.

```
;======================================================================
; Initialize standard I/O
;----------------------------------------------------------------------
        .glb     _init
        jsr.a    _init

;======================================================================
; Call main() function
```

Figure 2.19 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.

*1. The init function also initializes the microcomputer (hardware) for standard in-put/output functions. By default, the M16C/62 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 *_mbase
(2)Initialization of external variable char *_mnext
Initializes the heap_top label, which is the starting address of the heap section
(3)Initialization of external variable unsigned_msize
Initializes the "HEAPSIZE" expression, which sets at "2.2.2 e heap section size".

Figure 2.16 shows the initialization performed in ncrt0.a30.

```
;======================================================================
; heap area initialize
;-----------------------------------------------------------------------

    .glb  __mbase
    .glb  __mnext
    .glb  __msize
    mov.w #(heap_top&0FFFFH), __mbase
    mov.w #(heap_top>>16),__mbase+2
    mov.w #(heap_top&0FFFFH), __mnext
    mov.w #(heap_top>>16),__mnext+2
    mov.w #(HEAPSIZE&0FFFFH), __msize
    mov.w #(HEAPSIZE>>16),__msize+2
```

Figure 2.16 Initialization When Using Memory Management Functions (ncrt0.a30)

If you are not using the memory management functions, comment out the whole initialization section. 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:
[1]Call the C subroutine only after clearing them, B and D flags.
[2]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.[*1]

Stack size is calculated to use the stack size calculation utility STK Viewer & stk.

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.

```
;---------------------------------------------------------------------
; HEAP SIZE definition
;---------------------------------------------------------------------
HEAPSIZE        .equ    300h
```

Figure 2.21 Example of Setting Heap Section Size (ncrt0.a30)

f. Setting the interrupt vector table

Set the top address of the interrupt vector table to the part of Figure 2.22 in ncrt0.a30. The INTB Register is initialized by the top address of the interrupt vector table.

```
;---------------------------------------------------------------------
; INTERRUPT VECTOR ADDRESS  definition
;---------------------------------------------------------------------
VECTER_ADR        .equ    0ffd00h
```

Figure 2.22 Example of Setting Top Address of Interrupt Vector Table (ncrt0.a30)

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.

*1. 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.

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 2.23.

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

Figure 2.23 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 cross-environment compilers such as NC30, the user must determine the memory mapping.

With NC30, 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 :

Table 2.12 Section Names

Section Base Name _ Attribute

Table 2.13 shows Section Base Name and Table 2.14 shows Attributes.

Table 2.13 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 2.14 Section Naming Rules

Attribute	Meaning	Target section base name
I	Section containing initial values of data	data
N/F/S	N...near attribute [*1]	data, bss, rom
	F...far attribute [*1]	
	S...SBDATA attribute	data, bss
E/O	E...Even data size	data, bss, rom
	O...Odd data size	

*1.near and far are NC30 modifiers, used to clarify the addressing mode.
near.........accessible from 000000H to 00FFFFH
far...........accessible from 000000H to 0FFFFFFH

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

Table 2.15 Section Names

Section name	Contents
stack	This area is used as a stack. Allocate this area at addresses between 0400H to 0FFFFH.
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.
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.
fvector	This section stores the contents of the Micro Processor's fixed vector.
program	Stores programs
program_S	Stores programs for which #pragma SPECIAL has been specified.
switch_table	The section to which the branch table for switch statements is allocated. This section is generated only with the -fSOS option.

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 2.24 shows the how the sections are mapped according to the sample startup program's include file sect30.inc.

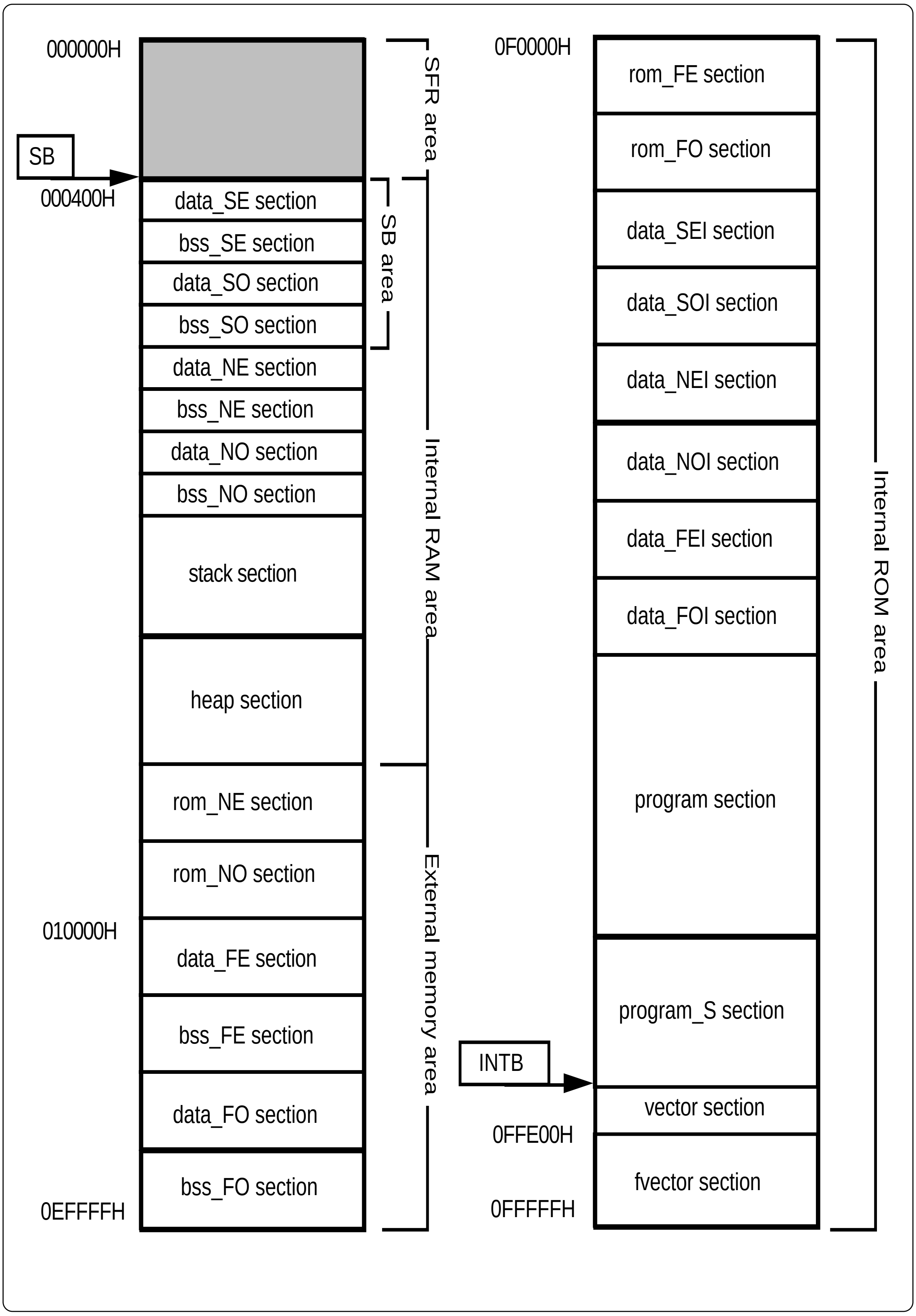


Figure 2.24 Example Section Mapping

b. Outline of memory mapping setup file

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

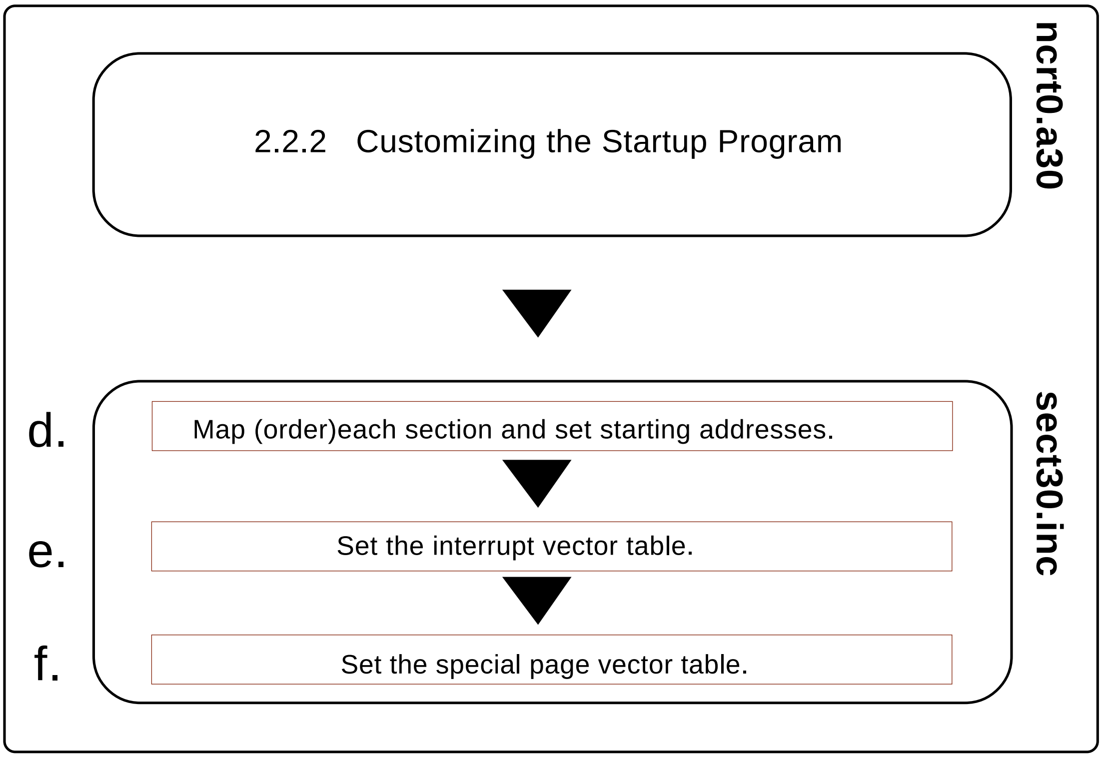


Figure 2.21 Example Sequence for Modifying Startup Programs

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 as30 pseudo instruction .ORG to specify their starting addresses. Figure 2.22 is an example of these settings.

```
        .section        program
        .ORG            0FFE000H        ⇐Specifies the starting address of the program section
;
```

Figure 2.22 Example Setting of Section Starting Address (sect30. inc)

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.

(a)Sections mapped to RAM

- data_SE section
- data_SO section
- data_NE section
- data_NO section
- data_FE section
- data_FO section
- bss_SE section
- bss_SO section
- bss_NE section
- bss_NO section
- bss_FE section
- bss_FO section

- stack section
- heap section

(b)Sections mapped to ROM

- rom_NE section
- rom_NO section
- rom_FE section
- rom_FO section

- program section
- interrupt section
- fvector section

- data_SEI section
- data_SOI section
- data_NEI section
- data_NOI section
- data_FEI section
- data_FOI section
- switch_table section

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

(a)Sections mapped only to 0H - 0FFFFH(near area)

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

(b)Sections mapped only to 0F0000H - 0FFFFFH

- program_S
- fvector

(c)Sections mapped to any area for the M16C/60 series

- data_FE section
- data_FO 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
- bss_FE section
- bss_FO section
- program
- vector

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

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

(2) Example Section Mapping in Single-Chip Mode

Figures 2.27, 2.28 and 2.29 are examples of the sect30.inc include file which is used for mapping sections to memory in single-chip mode.

```
;*****************************************************************************
;
; C Compiler for M16C/60,20
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
; Written by X.XXXXXXXX
;
; sect30.inc     : section definition
; This program is applicable when using the basic I/O library
;
; $Id: sect30.inc,v 1.14 2003/03/27 10:57:43 XXXXXXXX Exp $
;
;*****************************************************************************
;---------------------------------------------------------------
;
;   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   stack,DATA
    .blkb STACKSIZE
stack_top:

    .blkb ISTACKSIZE
istack_top:


;---------------------------------------------------------------
;   heap section
;---------------------------------------------------------------
    .section   heap,DATA
heap_top:
    .blkb HEAPSIZE
```

Figure 2.27 Listing of sect30.inc in Single-Chip Mode (1/3)

```
;---------------------------------------------------------------
; Near ROM data area
;---------------------------------------------------------------
    .section   rom_NE,ROMDATA,ALIGN
rom_NE_top:

    .section   rom_NO,ROMDATA
rom_NO_top:

;---------------------------------------------------------------
; Far RAM data area
;---------------------------------------------------------------
    .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:
```

⇐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.

```
;---------------------------------------------------------------
; Far ROM data area
;---------------------------------------------------------------
    .section   rom_FE,ROMDATA
    .org       0F0000H
rom_FE_top:

    .section   rom_FO,ROMDATA
rom_FO_top:

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

    .section   data_SOI,ROMDATA
data_SOI_top:

    .section   data_NEI,ROMDATA
data_NEI_top:

    .section   data_NOI,ROMDATA
data_NOI_top:

    .section   data_FEI,ROMDATA
data_FEI_top:

    .section   data_FOI,ROMDATA
data_FOI_top:

;---------------------------------------------------------------
; Switch Table Section
;---------------------------------------------------------------
    .section        switch_table,ROMDATA
switch_table_top:


;---------------------------------------------------------------
; code area
;---------------------------------------------------------------

    .section   program

    .section   interrupt
    ;.org ;must be set internal ROM area
    .section   program_S
```

Figure 2.28 Listing of sect30.inc in Single-Chip Mode (2/3)

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

.if M62TYPE==1
    .lword     dummy_int       ; BRK (vector 0)
    .lword     dummy_int       ;     (vector 1)
    .lword     dummy_int       ;     (vector 2)
    .lword     dummy_int       ;     (vector 3)
    .lword     dummy_int       ; int3(for user)(vector 4)
       :
       :
    (omitted)
       :
       :
    .lword     dummy_int       ; vector 62
    .lword     dummy_int       ; vector 63

;===============================================================
; fixed vector section
;---------------------------------------------------------------
    .section   fvector               ; fixed vector table
;===============================================================
; special page defination
;---------------------------------------------------------------
;   macro is defined in ncrt0.a30
;   Format: SPECIAL number
;
;---------------------------------------------------------------
;   SPECIAL 255
;   SPECIAL 254
       :
       :
    (omitted)
       :
       :
;   SPECIAL 19
;   SPECIAL 18
;
;===============================================================
; fixed vector section
;---------------------------------------------------------------
    .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
;
;*****************************************************************************
;
;   C Compiler for M16C/60,20
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;*****************************************************************************
```

Figure 2.29 Listing of sect30.inc in Single-Chip Mode (3/3)

e. Setting Interrupt Vector Table

If your program uses interrupt processing, change the interrupt vector table table in the vector section of sect30.inc. Figure 2.30 is an example of an interrupt vector address table.

```
;---------------------------------------------------------------
; variable vector section
;---------------------------------------------------------------
.section   vector       ; variable vector table
.orgVECTOR_ADR

.lword     dummy_int          ; vector 0 (BRK)              ⇐BRK instruction
.org(VECTOR_ADR +44)
.lword     dummy_int    ; DMA0 (for user)                    ⇐DMA0 interrupt
.lword     dummy_int    ; DMA1 (for user)                    ⇐DMA1 interrupt
.lword     dummy_int    ; input key (for user)               ⇐key input interrupt
.lword     dummy_int    ; AD Convert (for user)              ⇐ADC interrupt
.org       (VECTOR_ADR+68)
.lword     dummy_int    ; uart0 trance (for user)            ⇐UART0 send interrupt
.lword     dummy_int    ; uart0 receive (for user)           ⇐UART0 receive interrupt
.lword     dummy_int    ; uart1 trance (for user)            ⇐UART1 send interrupt
.lword     dummy_int    ; uart1 receive (for user)           ⇐UART1 receive interrupt
.lword     dummy_int    ; TIMER A0 (for user)                ⇐Timer A0 interrupt
.lword     dummy_int    ; TIMER A1 (for user)                ⇐Timer A1 interrupt
.lword     dummy_int    ; TIMER A2 (for user)                ⇐Timer A2 interrupt
.lword     dummy_int    ; TIMER A3 (for user)                ⇐Timer A3 interrupt
.lword     dummy_int    ; TIMER A4 (for user)(vector 25)     ⇐Timer A4 interrupt
.lword     dummy_int    ; TIMER B0 (for user)(vector 26)     ⇐Timer B0 interrupt
.lword     dummy_int    ; TIMER B1 (for user)(vector 27)     ⇐Timer B1 interrupt
.lword     dummy_int    ; TIMER B2 (for user)(vector 28)     ⇐Timer B2 interrupt
.lword     dummy_int    ; INT0 (for user)(vector 29)         ⇐External interrupt INT0
.lword     dummy_int    ; INT1 (for user)(vector 30)         ⇐External interrupt INT1
.lword     dummy_int    ; INT2 (for user)(vector 31)         ⇐External interrupt INT2
```

* dummy_int is a dummy interrupt processing function.

Figure 2.30 Interrupt Vector Address Table (sect30.inc)

The contents of the interrupt vectors varies according to the machine in the M16C/60 series. 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. 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.

[2] 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 2.31 is an example of registering the UART1 send interrupt processing function uarttrn.

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

    (omitted)
```

Figure 2.31 Example Setting of Interrupt Vector Addresses (sect30.inc)

f. Setting SPECIAL Page Vector Table

When using #pragma SPECIAL, use sect30.inc to set the special page vector table. Figure 2.32 is an example of setting the special page vector table.

```
;===============================================================
; special page definition
;---------------------------------------------------------------
;       macro is defined in ncrt0.a30
;       Format: SPECIAL number
;
;---------------------------------------------------------------
;
        SPECIAL 42
        SPECIAL 41
        SPECIAL 40
;
        SPECIAL 31
        SPECIAL 30
;
        SPECIAL 22
        SPECIAL 21
        SPECIAL 20
        SPECIAL 19
        SPECIAL 18
;
```

Figure 2.32 Example Setting of Special Page Vector Table

By default, the special page vector table is a comment. "SPECIAL" is a macro, whose behavior is associated with the function name defined by "#pragma SPECIAL".

To define a special page number you want to use, remove the comment for the desired page number.

Special page numbers do not need to be consecutive, but must always be set in descending order.

Chapter 3
Programming Technique

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

3.1 Notes

Renesas Technology 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 Technology 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-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.

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 case, write the instruction directly in the program using the ASM function.

In this compiler, the instructions which cannot be used may be generated for writing and read-out to the register of SFR area.

When accessing registers in the SFR area in C language, make sure that the same correct instructions are generated as done by using asm functions, regardless of the compiler's version and of whether optimizing options are used or not.

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.

[Example: C language description to SFR area]

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

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

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

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;
funC()
{
   port;
}
```

Figure 3.1 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 3.2.

```
extern int volatile port;
funC()
{
   port;
}
```

Figure 3.2 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 3.3 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 representable 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 3.4 Program Not Executed

(4) Operation between constants

Operation between constants is performed when compiling.

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

Figure 3.5 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 3.6 Example of Ambiguously Interpreted volatile Qualifier Statements

For successive bit manipulations, if optimized, the compiler generates codes to perform bit manipulations collectively, even when the volatile qualifier is specified. (Bit manipulations are performed simultaneously by overriding the order of references.)

To inhibit collective bit manipulations, use the "-Ono_bit (shortcut -ONB)" option.

3.1.4 Precautions on Using register Variables

a. register qualification and "-fenable_register" option

f the 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.

3.1.5 About Startup Handling

Startup may need to be modified depending on the type of microcomputer you are using or depending on your application system.

For modifications pertinent to the type of microcomputer, consult the data book, etc. for your microcomputer and correct the startup file included with the compiler package before use.

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 judgements.

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 64 Kbytes
Subscripts are calculated in 16-bit width. This ensures efficient access for arrays of 64 Kbytes or less in size.

[2]When the array size is greater than 64 Kbytes or unknown
Subscripts are calculated in 32-bit width.

Therefore, when it is known that the array size does not exceed 64 Kbytes, explicitly state the size in extern declaration of far type array as shown in Figure 3.7 or add the -fsmall_array (-fSA)*2 option 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 3.7 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 -fsmall_array (-fSA) option is specified, the compiler assumes an array of an unknown size to be within 64 Kbytes as it generates code.In the entry version, this option cannot be specified.

c. Array Subscripts

Array subscripts are type-extended during operations according to the size of each element in the array.

[1]2 bytes or more (other than char or signed char types)
Subscripts are always extended to int types for operations.
[2]far arrays of 64KB or more
Subscripts are always extended to long types for operations.

Therefore, if you declare variables that will be array subscripts as char types, they will be extended to int types each time they are referenced and therefore the code will not be efficient. In such cases, declare variables that will be array subscripts as int types.

d. 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 placed 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 pushing onto 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.

e. 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 writing the description shown in Figure 3.8.

* This Compiler assumes as a precondition that the SB register is initialized after a reset, and that it thereafter is used as a fixed register.

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

Figure 3.8 Example of variable declaration using SB-based addressing mode

f. Compressing ROM Size Using 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*".

g. 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 3.9 does not initialize the variable, which must therefore be initialized to 0 (by clearing the bss[*1] area) during the startup routine.

```
static int i;
```

Figure 3.9 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
;---------------------------------------------------------------------
;       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

                :
            (omitted)
                :
;=====================================================================
; FAR area initialize.
;----------------------------------------------------------------------
; bss zero clear
;----------------------------------------------------------------------
;       BZERO   bss_FE_top,bss_FE
;       BZERO   bss_FO_top,bss_FO
```

Figure 3.10 Commenting Out Routine to Clear bss Area

*1. 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 3.11 is an example of calling assembler function asm_func.

```
extern void near        asm_func( void );          ⇐Assembler function
                                                      prototype declaration
void main()
{
                :
                 (omitted)
                :
        asm_func();          ⇐Calls assembler function
}
```

Figure 3.11 Example of Calling Assembler Function Without Parameters(smp1.c)

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

        rts
```

Figure 3.12 Compiled result of smp1.c(smp1.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,A1A0), 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 3.13 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
void main()                                   assembler function.
{
        int         i = 0x02;
        int         j = 0x05;

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

Figure 3.13 Example of Calling Assembler Function With Parameters (smp2.c)

```
        .glb    _main
_main:
        enter   #04H

        mov.w   #0002H,-4[FB]   ;  i
        mov.w   #0005H,-2[FB]   ;  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       ⇐Calls assembler function(preceded by '_')

        exitd
```

Figure 3.14 Compiled result of smp2.c(smp2.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 (float and double) parameters

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. Sections can be assigned any desired name.
[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.*1

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.*1
[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 3.15 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]
    PUSHM        R3,R1               ⇐[5]
    MOV.w          SYM1, R3
    MOV.w       SYM1+2,R1
                (omitted)
                :
    POPM        R3,R1                ⇐[6]
    POPC        FLG                  ⇐[7]
    RTS                              ⇐[8]
    .END
```
* [1] to [8] correspond to the steps described above.

Figure 3.15 Example Coding of Assembler Function

*1.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 3.16 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.
Compound 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             #01000H, R2      ;32-bit data
          MOV.W             #0A00H,R0
          FCLER             D, B
          FSET              U
          RTS
          .END
```

Figure 3.16 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 3.17 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

main()
{
                :
             (omitted)
                :
}
```

[Assembler function]

```
        .GLB            _counter        ⇐External declaration of C program's
_asm_func:                                global variable
                :
             (omitted)
                :
        MOV.W           _counter, R0    ⇐Reference
```

Figure 3.17 Referencing a C Global Variable

d. Notes on Coding Interrupt Handling 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 3.18 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            ⇐Push registers.

        MOV.B   #01H, R0L
               :
             (omitted)
                :
        popm    R0,R1,R2,R3,A0,A1,FB            ⇐Pull registers.
        reit                                    ⇐Return to C program
        .END
```

Figure 3.18 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) Make sure the registers used in the assembler functions are saved before calling any C language function, and that they are restored after returning from the C language function.

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. Therefore, do not modify the FB value in assembler functions. If it is yet necessary to modify the FB register for reason of system design, save it to the stack at the beginning of a function and restore it when returning to the function from which it 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 14.5 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 3.16 Example Coding of Assembler Function

#pragma PARAMETER passes arguments to assembler functions via 16-bit general-purpose 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

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 Mitsubishi C compilers "NCxxx" 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
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

% nc30Δ[*command-line-option*]Δ[*assembly-language-source-file-name*]Δ
[*relocatable-object-file-name*]Δ<*C-source-file-name*>

%	: Prompt
< >	: Mandatory item
[ ]	: Optional item
Δ	: 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 file (extension .r30) and ends processing [*1]
-D*identifier*	Defines an identifier. Same function as #define.
-I*directory*	Specifies the directory containing the file(s) specified in #include. You can specify up to 8 directories.
-E	Invokes only preprocess commands and outputs result to standard output.[*1]
-P	Invokes only preprocess commands and creates a file (extension .i). [*1]
-S	Creates an assembly language source file (extension .a30) and ends processing.[*1]
-U *predefined macro*	Undefines the specified predefined macro.
-silent	Suppresses the copyright message display at startup.
-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).

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.

-c

Compile driver control

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

Execution example :

```
%nc30 -c sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c

% ls sample.*
-rw-r--r--  1 toolusr      2835 Aug 17 11:28 sample.c
-rw-r-----  1 toolusr       450 Aug 17 11:28 sample.r30
%
```

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

-D*identifier*

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.

Execution example :

```
%nc30 -c -DMYDEBUG=1 -DMSDOS=1 -DUNIX sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c

%
```

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.*

-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 8 directory.

Syntax : nc30Δ-I *directory*Δ<C source file>

Execution example :

```
% nc30 -c -I./test/include -I./test/inc sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c

%
```

* In this example, two directories, ./test/include and ./test/inc are specified.

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.

-E

Compile driver control

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

Execution example :

```
% nc30 -E sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

#line 1 "sample.c"
          :
      (omitted)
          :
#line 1 "/usr3/tool/toolusr/work30/inc30/stdio.h"
          :
      (omitted)
          :
```

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

-P

Compile driver control

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

Execution example :

```
% nc30 -P sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
%ls sample.*
-rw-r--r--  1 toolusr      2835 Aug 17 11:28 sample.c
-rw-r-----  1 toolusr      2322 Aug 17 11:30 sample.i
%
```

Notes :

1. When this option is specified, no assembly source file (extensions .a30), relocatable object 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

Execution example :

```
% nc30 -S sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
% ls sample.*
-rw-r-----  1 toolusr      2059 Aug 17 11:30 sample.a30
-rw-r--r--  1 toolusr      2835 Aug 17 11:28 sample.c
%
```

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

-U*predefined macro*

Compile driver control

Function : Undefines predefined macro constants

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

Execution example :

```
% nc30 -c -UNC30 -UM16C sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c

%
```

*In this example, macro definitions NC30 and M16C are undefined.

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.

-silent

Compile driver control

Function : Suppresses the display of copyright notices at startup

Execution example :

```
% nc30 -c -silent sample.c
sample.c

%
```

-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 : When the -S option is used, the -dsouce option is automatically enabled.
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" function, generates an assembly language list file (.lst).

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
-o*filename*	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	Specifies the destination directory of the file(s) (absolute module file, map file, etc.) generated by ln30.

-o *filename*

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 *filename*Δ<C source file>

Execution example :

```
% nc30 -o./test/sample ncrt0.a30 sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.a30
sample.c
% cd test
% ls
total 65
drwxr-x---   2 toolusr       512 Aug 17 16:13 ./
drwxrwxrwx  11 toolusr      3584 Aug 17 16:14 ../
-rw-r-----   1 toolusr     44040 Aug 17 16:14 sample.x30

%
```

* In this example, the option is used to specify that sample.x30, are output to directory ./test.

-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

Execution example :

```
% nc30 -dir./test/sample -o ncrt0.a30 sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.a30
sample.c
% cd test/sample
% ls
total 65
drwxr-x---  2 toolusr      512 Aug 17 16:13 ./
drwxrwxrwx 11 toolusr     3584 Aug 17 16:14 ../
-rw-r-----  1 toolusr    44040 Aug 17 16:14 ncrt0.a30

%
```

* In this example, the option is used to specify that ncrt0.a30, are output to directory ./test/sample.

Note : 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.

A.2.3 Version Information Display Option

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

Table 2.3   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

Execution example :

```
% nc30 -c -v sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
cpp30 sample.c -o sample.i -DM16C -DNC30
ccom30 sample.i -o ./sample.a30
as30 -. -N sample.a30

%
```

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

Execution example :

```
D:\MTOOL\nc30wa>nc30 -V
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

M16C/60 C Compile Driver        Version X.XX.XX
NC Preprocessor                 Version X.XX.XX
M16C/60 C Compiler              Version X.XX.XX
 Assembler Optimizer (aopt30) for M16C Family Version X.XX.XX
M16C/60 Series Assembler system Version X.XX ReleaseX
   Assembler Driver (as30) for M16C/60 Series Version X.XX.XX
   Macro Processor (mac30) for M16C/60 Version X.XX.XX
   Structured Processor (pre30) for M16C Family Version X.XX.XX
   Assembler Processor (asp30) for M16C Family Version X.XX.XX
   Linkage Editor (ln30) for M16C Family Version X.XX.XX
   Librarian (lb30) for M16C Family Version X.XX.XX
   Load Module Converter (lmc30) for M16C/60 Series Version X.XX.XX
   Cross Referencer (xrf30) for M16C Family Version X.XX.XX
   Absolute Lister (abs30) for M16C Family Version X.XX.XX

D:\MTOOL\nc30wa>
```

Supplement : Use this option to check that the compiler has been installed correctly. The 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 "M3T-NC30WA Guide" 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. In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-gno_reg	Suppresses the output of debugging information for register variables. In the entry version, this option cannot be specified.
-gold	outputs debugging information for old version debuggers and third-party debuggers . In the entry version, this option cannot be specified.

-g

Outputting debugging information

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

Execution example :

```
% nc30 -g -v sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
cpp30 sample.c -o sample.i -DM16C -DNC30
ccom30 sample.i -o ./sample.a30 -g

as30 -. -N --N sample.a30

ln30 sample.r30 -. -G -MS -o sample
          :
      (omitted)
          :
% ls sample.*
-rw-r--r--   1 toolusr      2894 Aug 17 14:51 sample.c
-rw-r-----   1 toolusr      7048 Aug 17 15:53 sample.map
-rw-r-----   1 toolusr     53570 Aug 17 15:53 sample.x30
%
```

Note : 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.

-genter

Outputting enter instruction

Function : Always output an enter instruction when calling a function.
In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.

Note: When using the debugger's stack trace function, always specify this option. Without this option, you cannot obtain the correct result.
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.
In the entry version, this option cannot be specified.

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.

-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.
In the entry version, this option cannot be 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:

PDB30 V.2.00 and later
XDB30 V.2.00 and later
PDB30SIM V.2.00 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.

In the entry version, all optimization options cannot be specified.

Table A.5 Optimization Options

Option	Short form	Function
-O[1-5]	None.	Effects the best possible optimization both in execution speed and in ROM capacity level by level
-OR	None.	Maximum optimization of ROM size followed by speed
-OS	None.	Maximum optimization of speed followed by ROM size
-Oconst	-OC	Performs optimization by replacing references to the const-qualified external variables with constants
-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_stdlib	-ONS	Inhibits inline padding of standard library functions and modification of library functions.
-Osp_adjust	-OSA	Optimizes code generation by combining stack correction codes after function calls. This helps to reduce the ROM capacity, as well as speed up processing. However, the amount of stack used may increase.
-Ostack_frame_align	-OSFA	Aligns the stack frame on an even boundary.
-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_logical _or_combine	-ONLOC	Suppresses the optimization that puts consecutive ORs together.
-Ono_asmpot	-ONA	Inhibits starting the assembler optimizer "aopt30."
-Ostatic_to_inline	-OSTI	A static function is treated as an inline function.

[Effect of each Optimization Options]

Option	-O	-OR	-OS	-OSA	-OSFA
SPEED	faster	lower	faster	faster	faster
ROM size	decrease.	decrease	increase	decrease.	same
usage of stack	decrease	same	same	increase	increase

-O[1-5]

Optimization

Function : Optimizes speed and ROM size to the maximum. This option can be specified with -g options.-O3 is assumed if you specify no numeric(no level)
In the entry version, this option cannot be specified.

-O1:Makes -O3,-Ono_bit,-Ono_break_source_debug,-Ono_float_const_fold, and -Ono_stdlib valid
-O2: Makes no diffrence with -O1
-O3:Optimizes speed and ROM size to the maximum.
-O4:Makes -O3 and -Oconst valid
-O5:Effect the best possible optimization in common subexpressions (if the option -OR is concurrentlyspecofied); effects the best possible optimization in transfer and comparison of character strings (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.

```
Exsample)

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

test1(){
    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 = 10)
```

The next page is followed.

-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.

[For examplr: C sauce which must not use an optimization option at the time of compile]

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

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

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

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.

-OR

Optimization

Function : Optimizes ROM size in preference to speed. This option can be specified with -g and -O options.
In the entry version, this option cannot be specified.

Supplement : 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.
In the entry version, this option cannot be specified.

-Oconst　　　　　　　　　　　　　　　　　　　　　　　　-OC

Optimization

Function : Performs optimization by replacing references to the const-qualified external variables with constants.This option is effective also at the time of the specification more than "-O4" option.
In the entry version, this option cannot be specified.

Supplement : Optimization is performed when the following conditions are satisfied simultaneously :

1. Extern variables excluding structures, unions, and arrays;
2. Extern variables declared using the const qualifier;
3. Extern variables initialized in the same C source file.

The following example shows code that can be optimized.

Code example :

```
int const i = 10;

func()
{
    int k = i;   /* i is replaced with 10. */
          :
          :

}
```

-Ono_bit　　　　　　　　　　　　　　　　　　　　　　　　-ONB

Suppression of optimization

Function : Suppresses optimization based on grouping of bit manipulations.
In the entry version, this option cannot be specified.

Supplement : When you specify -O (or -OR or -OS), 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 optimization is performed, The variables is specified regardless volatile-qualified.
- This option is only valid if you specify option -O[3 to 5] (or -OR or -OS).

-Ono_break_source_debug -ONBSD

Suppression of optimization

Function : Suppresses optimization that affects source line data.
In the entry version, this option cannot be specified.

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 valid only when the -OR or -O option is specified.

-Ono_float_const_fold -ONFCF

Suppression of optimization

Function : Suppresses the constant folding processing of floating point numbers.
In the entry version, this option cannot be specified.

Supplement : By default, NC308 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_stdlib -ONS

Suppression of optimization

Function : Suppresses inline padding of standard library functions, modification of library functions, and similar other optimization processing.
In the entry version, this option cannot be specified.

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.
In the entry version, this option cannot be specified.

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.

```
int func18 int, int );
int func2( int );

void main( void ) {
    int i = 1;
    int j - 2;
    int k;
    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.

-Ostack_frame_align -OSFA

Aligns stack frame

Function : Aligns the stack frame on an even boudary.
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 apecified in #pragma are not aligned.

#pragma INTHANDLER
#pragma HANDLER
#pragma ALMHANDLER
#pragma CYCHANDLER
#pragma INTERRUPT

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.All files should be compiled using this option.

-Oloop_unroll = [loop count] -OLU

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.
In the entry version, this option cannot be specified.

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.

1. 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.

-Ono_logical_or_combine -ONLOC

Suppression of optimization

Function : Suppresses the optinization that puts consective ORs together.
In the entry version, this option cannot be specified.

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 & 0x0 || 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 optimiza

-Ono_asmopt -ONA

Inhibits starting the assembler optimizer

Function : Inhibits starting the assembler optimizer "aopt30".
In the entry version, this option cannot be specified.

-Ostatic_to_inline   -OSTI

Optimization

Function : A static function is treated as an inline function and the assembling code which carried out inline deployment is generated.
In the entry version, this option cannot be specified.

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 satic function.
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.)

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().

Notle :

- The assembler code to description of substance of the static function which became inline function treatment is always generated.
- About a function, it is compulsorily. In treating as an inline function, it is in a function. Please make an inline declaration.

A.2.6 Generated Code Modification Options

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

Table A.6(1/2) Generated Code Modification Options

Option	Short form	Description
-fansi	None.	Makes -fnot_reserve_far_and_near, -fnot_reserve_asm, and -fextend_to_int valid. In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fnot_reserve_asm	-fNRA	Exclude asm from reserved words. (Only _asm is valid.) In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fnot_reserve_far_and_near	-fNRFAN	Exclude far and near from reserved words. (Only _far and _near are valid.) In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fnot_reserve_inline	-fNRI	Exclude far and near from reserved words. (Only _inline is made a reserved word.) In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fextend_to_int	-fETI	Performs operation after extending char-type data to the int type. (Extended according to ANSI standards.)*1 In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.
-fchar_enumerator	-fCE	Handles the enumerator type as an unsigned char type, not as an int type.
-fno_even	-fNE	Allocate all data to the odd section , with no separating odd data from even data when outputting .
-ffar_RAM	-fFRAM	Changes the default attribute of RAM data to far.
-fnear_ROM	-fNROM	Changes the default attribute of ROM data to near. In the entry version, this option cannot be specified.
-fconst_not_ROM	-fCNR	Does not handle the types specified by const as ROM data.

*1. char-type data or signed char-type data evaluated under ANSI rules is always extended to int-type 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.6(2/2) Generated Code Modification Options

Option	Short form	Description
-fnot_address_volatile	-fNAV	Does not regard the variables specified by #pragma ADDRESS (#pragma EQU) as those specified by volatile.
-fsmall_array	-fSA	When referencing a far-type array, this option calculates subscripts in 16 bits if the total size of the array is within 64K bytes. In the entry version, this option cannot be specified.
-fenable_register	-fER	Make register storage class available
-fno_align	-fNA	Does not align the start address of the function. In the entry version, this option cannot be specified.
-fJSRW	None.	Changes the default instruction for calling functions to JSR.W.
-fbit	-fB	Outputs a 1-bit manipulate instruction to all external variables arranged in 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 byte.
-fuse_DIV	-fUD	This option changes generated code for divide operation. In the entry version, this option cannot be specified.
-finfo	None.	Outputs the information required for the Inspector, STK Viewer, Map Viewer, and utl30. In the entry version, this option cannot be specified.
-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.

-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 .... Extends char-type data to int-type data to perform operations

In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.

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

-fnot_reserve_asm -fNRA

Modify generated code

Function : Removes asm from the list of reserved words. However, _asm, which has the same function, remains as a reserved word.
In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.

-fnot_reserve_far_and_near   -fNRFAN

Modify generated code

Function : Removes far and near from list of reserved words. However, _far and _near, which have the same functions, remain reserved words.
In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.

-fnot_reserve_inline   -fNRI

Modify generated code

Function : Does not handle inline as a reserved word. However, _inline that has the same function is handled as a reserved word.
In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by 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)
In the entry version, this option is always enabled (i.e., assumed to be specified). Therefore, it cannot be enabled or disabled by specification.

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 thechar type overflows in the middle of operation, and that the result takes on an unexpected value. An example is shown below.

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

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

In this case, thechar 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.

-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.

-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.

-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 (64-Kbyte area).

-fnear_ROM -fNROM

Modify generated code

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

In the entry version, this option cannot be specified.

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, however.

-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, however

-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. You do not normally need to use this option, however.

-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 64 Kbytes.

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 64 Kbytes 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 Kbytes 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 64 Kbytes.

-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:

1. Integral type variable
2. Pointer variable

Note : 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.

-fno_align -fNA

Changes generated code

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

In the entry version, this option cannot be specified.

-fJSRW

Changes 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. Conversely, if a function is called that is located 32 Kbytes 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.
This option is useful when the program is relatively small not exceeding 32 Kbytes in size or ROM compression is desired.

-fbit -fB

Modify generated code

Function : Outputs a 1-bit manipulate instruction to used an absolute addressing to all external variables arranged in the near area.

Supplement : If a external variable to perform bit manipulation resides in a 0000_{16} to $1FFF_{16}$ area of the M16C's memory space, the code efficiency generated by the compiler can be increased by specifying this option.

In application programming for the single chip, this option is effective if the RAM area is defined within the area as described above.
An error occurs at link-time when you use the argument other than above.

-fno_carry -fNC

Changes 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 64-Kbyte boundary. As a result, the code will be more efficient.

Note : 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 64-Kbyte boundary.

-fauto_128 -fA1

Changes generated code

Function : Limits the usable stack frame to 128 bytes

-fuse_DIV -fUD

Changes generated code

Function : This option changes generated code for divide operation.
In the entry version, this option cannot be specified.

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.

Note : If the divide operation results in an overflow when this option is specified, the compiler may operate differently than stipulated in ANSI.
The div instruction of the M16C 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.

-finfo

Changes generated code

Function : Outputs the information required for the TM, Inspector, STK Viewer, Map Viewer, and utl30.
In the entry version, this option cannot be specified.

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

Note : 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.

-fswitch_other_section — -fSOS

Changes 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’
This option does not normally need to be used.

-fchange_bank_always — -fCBA

Changes generated code

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

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

A.2.7 Library Specifying Option

Table A.7 lists the startup options you can use to specify a library file.

Table A.7 Library Specifying Option

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

-l*libraryfilename*

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>

Execution example :

```
% nc30 -v -lusrlib ncrt0.a30 sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.a30
as30 -. -N ncrt0.a30

sample.c
cpp30 sample.c -o sample.i -DM16C -DNC30
ccom30 sample.i -o ./sample.a30
main
as30 -. -N sample.a30


ln30 ncrt0.r30 sample.r30 -. -l usrlib -o ncrt0

%
```

*** In this example, the option is used to specify a library named "usrlib.lib."**

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. (If you specify multiple libraries, nc30lib.lib is given the lowest priority as it is referenced.)

A.2.8 Warning Options

Table A.8 shows the command line options for outputting warning messages for contraventions of nc30 language specifications.

Table A.8 Warning Options

Option	Short form	Function
-Wnon_prototype	-WNP	Outputs warning messages for functions without prototype declarations.
-Wunknown_pragma	-WUP	Outputs warning messages for non-supported #pragma.
-Wno_stop	-WNS	Prevents the compiler stopping when an error occurs.
-Wstdout	None.	Outputs error messages to the host machine's standard output (stdout).
-Werror_file<file name>	-WEF	Outputs error messages to the specified file.
-Wstop_at_warning	-WSAW	Stops the compiling process when a warning occurs.
-Wnesting_comment	-WNC	Outputs a warning for a comment including */ .
-Wccom_max_warnings = Warning Count	-WCMW	This option allows you to specify an upper limit for the number of warnings output by ccom30.
-Wall	None.	Displays message for all detectable warnings(however, not including alarms output by -Wlarge_to_small and -Wno_used_argument).
-Wmake_tagfile	-WMT	Outputs error messages to the tag file of source-file by source-file.
-Wuninitialize_variable	-WUV	Outputs a warning about auto variables that have not been initialized.
-Wlarge_to_small	-WLTS	Outputs a warning about the tacit transfer of variables in descending sequence of size.
-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.
-Wno_used_argument	-WNUA	Outputs a warning for unused argument of functions.

-Wnon_prototype   -WNP

Warning option

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.

-Wunknown_pragma -WUP

Warning option

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

-Wno_stop -WNS

Warning option

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.

Note : 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.

-Wstdout

Warning option

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

Execution example :

```
A> nc30 -c -Wstdout sample.c > err.doc

A> type err.doc
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
[Error(ccom):sample.c,line 39] unknown valuable port00
   ==⇒         port00 = 0x00;
Sorry, compilation terminated because of these errors in main().

A>
```

Supplement : Use this option to save error output, etc. to a file by using Redirect in the MS-Windows95 version (personal computer version).

Note : In this Compiler for MS-Windows version(personal computer version), errors from as30 and ln30 invoked by the compile-driver are output to the standard output regardless of this option.

-Werror_file <file name> -WEF

Warning option

Function : Outputs error messages to the specified file

Syntax : nc30Δ-Werror_fileΔ<output error message file name>

Supplement : 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.

Output example:

test.c12 Error(ccom):unknown variable i

-Wstop_at_warning -WSAW

Warning option

Function : When a warning occurs, the compiler's end code is set to "10" as it is returned.

Supplement : If a warning occurs when compiling, the compilation by default is terminated with the end code "1" (terminated normally).
Use this option when you are using the make utility, etc. and want to stop compile processing when a warning occurs.

-Wnesting_comment -WNC

Warning option

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

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

-Wccom_max_warnings =Warning Count　-WCMW

Warning option

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.

Note : 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.

-Wall

Warning option

Function : Displays message for all detectable warnings(however, not including alarms output by -Wlarge_to_small(-WLTS) and -Wno_used_argument(-WNUA),which are displayed with the -Wnon_prototype(-WNP) and -Wunknown_pragma (-WUP) options and in the following cases (1) and (2). Note that these warnings are not all coding errors
because they are the compiler's inference.

Case (1)
When the assignment operator = is used in the if statement, the for statement or a comparison statement with the && or || operator.

```
Example:    if( i = 0 )
                func();
```

Case (2)
When "==" is written to which '=' should be specified.

```
Example:    i == 0;
```

Case(3)
When function is defined in old format.

```
Example:    func(i)
            int  i;
            {
                  :
              (omitted)
                  :
            }
```

Note : 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.

-Wmake_tagfile -WMT

Warning option

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<file name>î(-WEF) option canít specify.

-Wuninitialize_variable -WUV

Warning option

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

Supplement : If an auto variable is initialized in conditional jump by, for example, a if 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.

```
Example)

main()
{
    int  i;
    int  val;
    for  (i =0;i<2;I++) {
          f();
          val =1 ;// Initalize  by logical
    }
    ff(val );
}
```

-Wlarge_to_small -WLTS

Warning option

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

- 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.

- 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

-Wno_warning_stdlib -WNWS

Warning option

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

-Wno_used_argument -WNUA

Warning option

Function : Outputs a warning for unused arguments function.

A.2.9 Assemble and Link Options

Table A.9 shows the command line options for specifying as30 and ln30 options.

Table A.9   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. In the entry version, this option cannot be specified.
-ln30Δ<*option*>	Specifies options for the ln30 assemble command. If you specify two or more options, enclose them in double quotes. In the entry version, this option cannot be specified.

-as30*"option"*

Assemble/link option

Function : Specifies as30 assemble command options

If you specify two or more options, enclose them in double quotes.

In the entry version, this option cannot be specified.

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

Execution example : In the example below, the assembler list file is generated when compiling.

```
% nc30 -v -as30 " -l -s " sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
cpp30 sample.c -o sample.i -DM16C -DNC30
ccom30 sample.i -o ./sample.a30
as30 -. -N -l -s sample.a30
% ls sample.*
-rw-r--r--  1 toolusr      2850 Aug 17 14:51 sample.c
-rw-r-----  1 toolusr     10508 Aug 17 15:43 sample.lst     ←*
-rw-r-----  1 toolusr       587 Aug 17 15:43 sample.r30
%
```

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

For reference, the following table lists the AS30 V.4.00 options.

Option	Description
-.	Inhibits all message outputs on the screen. Use this option if you do not want anything to be displayed on the screen when executing AS30 in a batch file, etc. * Do not specify this option in the option -as30.
-A	Evaluates mnemonic operand.
-C	Displays the content of a command line when as30 starts up mac30 and asp30. * Do not specify this option in the option -as30.
-D	Sets a constant to a symbol.
-F	Fixes the file name of ..FILE development to a source file name.
-H	Header information is not output to an assembler list file.
-I	The include file specified by ".INCLUDE" that is written in the source file is searched from a specified directory.
-L	Generates an assembler list file (extension .lst).
-M	Generates structured description command variables in byte type. * Do not specify this option in the option -as30.
-N	Does not output information on macro instruction lines.
-O	Specifies the output destination path for a generated file. A directory or drive name can be specified for the path. If this specification is omitted, the generated file is output to the same path as that of the source file. * Do not specify this option in the option -as30.
-P	Processes structured description command. * Do not specify this option in the option -as30.
-S	Outputs local symbol information.
-T	Generates an assembler error tag file.
-V	Displays the version of the assembler system program. * Do not specify this option in the option -as30.
-X	Starts up an external program using a tag file as argument. * Do not specify this option in the option -as30.

*You can specify the assembler's option to use option -as30, using nc30. In this case, 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.
In the entry version, this option cannot be specified.

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

Execution example : In the example below, the map file is generated when compiling.

```
% nc30 -g -v -osample -ln30 -ms ncrt0.a30 sample.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.a30
as30 -. -N --N ncrt0.a30

sample.c
cpp30 sample.c -o sample.i -DM16C -DNC30
ccom30 sample.i -o ./sample.a30 -g
as30 -. -N --N sample.a30


ln30 ncrt0.r30 sample.r30 -. -G -MS -ms -o sample
          :
      (omitted)
          :
% ls sample.*
-rw-r--r--  1 toolusr      2850 Aug 17 14:51 sample.c
-rw-r-----  1 toolusr     44040 Aug 17 15:47 sample.x30
-rw-r-----  1 toolusr      8310 Aug 17 15:47 sample.map     ⇐❋
%
```

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

For reference, the following table lists the options for ln30, which is part of the AS30 V.4.10 package.

Option	Description
-.	Inhibits all message outputs on the screen. Use this option if you do not want anything to be displayed on the screen when executing AS30 in a batch file, etc. * Do not specify this option in the option -ln30.
-E	Specifies the start address of an absolute object module.
-G	Outputs source debug information to an absolute file. * Do not specify this option.
-L	Specifies the library file name to be referenced.
-LD	Specifies the directory of the library to be referenced.
-LOC	This command option outputs the data of a specified section to an absolute file beginning with a specified address. However, symbol values (addresses), etc. within the section do not change.
-M	Generates a map file.
-MS	Generates a map file including symbol information.
-MSL	The fullname of symbol more than 16 characters are output to mapfile(xx.map).
-NOSTOP	Outputs all encountered errors to the screen.
-O	Specifies an absolute file name. * Do not specify this option in the option -ln30.
-ORDER	Specifies the addresses of sections and the order in which they are arranged. * Do not specify this option in the option -ln30.
-T	Outputs a link error tag file. * Do not specify this option in the option -ln30.
-V	Displays the version of the linkage editor * Do not specify this option in the option -ln30.
@file	Specifies command file. * Do not specify this option in the option -ln30.

*You can specify the assembler's option to use option -ln30, using nc30. In this case, 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 files.
- -S : Stop after creating assembly language source files.

Appendix B
Extended Functions Reference

To facilitate its use in systems using the M16C/60 series, 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 (1/2)

Extended feature	Description
near/far qualifiers	1. 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). Example 1 : `asm("   MOV.W    R0, $$[FB]",f);` Example 2 : `asm("   MOV.W    R0, $$",s);` Example 3 : `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.
Default argument declaration for function	1. Default value can be defined for the argument of a function. Example 1 : `extern int func(int=1, char=0);` Example 2 : `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	1. Functions can be inline developed by using the inline storage class specifier. Example : inline func(int i); * Always be sure to define the body of an inline function before using the inline function.

Table B.2   Extended Functions (2/2)

Extended feature	Description
Extension of Comments	1. 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/60 series in C language.
macro assebler function	You can describe some assembler command as the function of C language. Exampe : char dadd_b(char val1, char val2); Example : int dadd_w(int val1, int val2);

B.1 Near and far Modifiers

For the M16C/60,20 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 specifierΔnear or farΔvariable;
```

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;

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

Figure B.2 Example of Variable Declaration

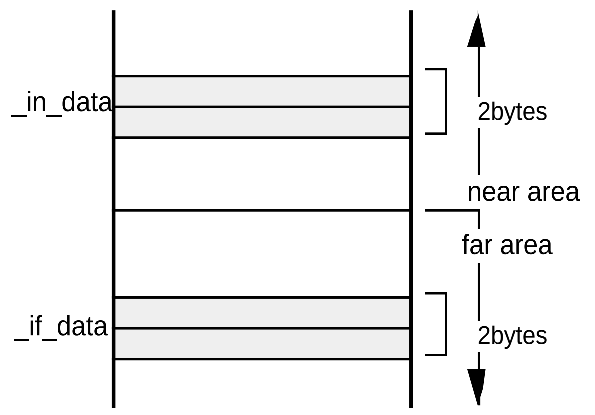


Figure B.3 Memory Location of Variable

B.1.3 Format of Pointer type Variable

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

● Example

```
int  *  ptr;
```

Figure B.4 Example of Declaring a Pointer Type Variable(1/2)

Because the variables are located near and take on the variable type far, 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(2/2)

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 above examples.

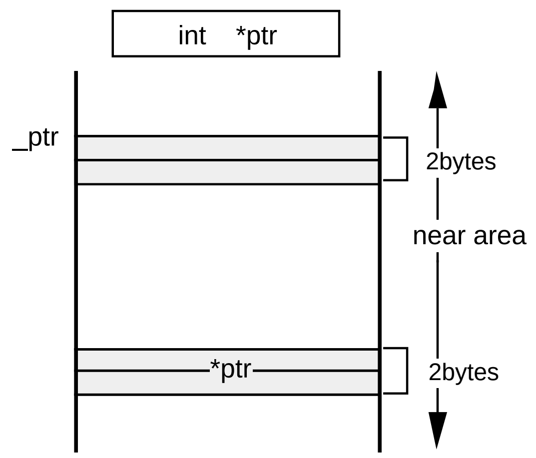


Figure B.6 Memory Location of Pointer type Variable

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

● Example 1

```
int   far   *ptr1;
```

● Example 2

```
int   *   far   ptr2;
```

Figure B.7 Example of Declaring a Pointer Type Variable(1/2)

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

● Example 1

```
int   far   *   near   ptr1;
```

● Example 2

```
int   near   *   far   ptr2;
```

Figure B.8 Example of Declaring a Pointer Type Variable(2/2)

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 2-byte variable that indicates the int-type variable located in the near area. The variable itself is located in the far area.

Memory mappings for Examples 1 and 2 are shown in Figure B.9.

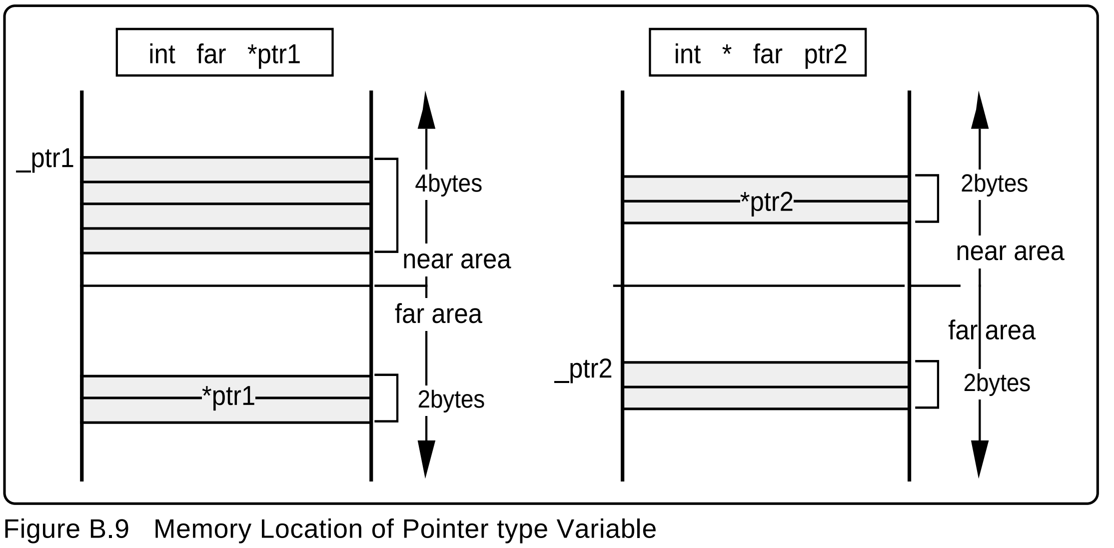


Figure B.9 Memory Location of Pointer type Variable

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 / far Control by nc30 Command Line Options

NC30 handles functions as belonging to the far attribute and variables (data) as belonging to the near attribute 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.1 nc30 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;

main()
{
    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;

main()
{
    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 continues processing by assuming the attribute of that function is far. Figure B.12 shows a display example where a function is declared to be near.

```
%nc30 -S smp.c
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION 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;

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

This declaration is interpreted as the following:

```
extern int far idata = 10;

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

Figure B.13 Integrated Function of Function 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 common header file.

common header file
common.h

```
extern int far          data;
```

C source file
a.c

```
#include   "common.h"
main()
{
      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. The asm function also has extended functions for manipulating the m and x flags and referencing variables written in C.

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
{
        :
    (abbreviated)
        :
     ta0_int = 0x07;                  ⇐Permits timer A0 interrupt
     asm("        FSETI");            ⇐Sets interrupt enable flag
}
```

Figure B.16 Example of Description of asm Function (1/2)

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(2/2)

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 destroy register contents in the 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 Description 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()
{
        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 FB offset

Figure B.20 shows an example for referencing an auto variable and its compile result.

● C source file

```
void func()
{
        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 shown in Figure B.21 so that auto variables in an asm function use a 1-bit 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 Bit Position

Figure B.23 shows examples of referencing auto area bit fields and the results 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    (       s)      size   1,       offset -1
;## #   ARG Size(0)     Auto Size(1)    Context Size(5)

        .section        program
        ._file  'bit.c'
        ._line  3
        .glb    _func
_func:
        enter   #01H
        ._line  11
;#### ASM START
        bset 1,-1[FB]   ;  s

;#### ASM END
        ._line  12
        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    $$",  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(3)


        .section        program
        ._file  'reg.c'
        ._line  3
;## # C_SRC :   {
        .glb    _func
_func:
        ._line  4
;## # C_SRC :           register int i=1;
        mov.w   #0001H,R0       ;  i
        ._line  6
;## # C_SRC :           asm("   mov.w   $$,A1",i);
;#### ASM START
        mov.w   R0,A1   ;  i            ⇐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 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    R1,   $$",  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()
{
             :
       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 FB offset

See Figure B.28 for examples of referencing extern and static variables.

● C source file

```
extern int ext_val;                          ⇐extern variable

func()
{
        static int s_val;                    ⇐static variable

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

● Assembly language source file(compile result)

```
        .glb    _func
_func:
        ._line  8
;## # C_SRC :         asm("   mov.w   #01H,$$",ext_val);
;#### ASM START
        mov.w   #01H,_ext_val                ⇐Move to _ext_val
        ._line  9
;## # C_SRC :         asm("   mov.w   #01H,$$",s_val);
        mov.w   #01H,___S0_s_val             ⇐Move to ___S0_s_val

;#### ASM END
        ._line  12
;## # C_SRC :   }
        rts

        .SECTION        bss_NE,DATA
___S0_s_val:            ;### C's name is s_val
        .blkb   2
        .glb    _ext_val
_ext_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(3)

        .section        program
        ._file  'kk.c'
        ._line  10
;## # C_SRC :   {
        .glb    _func
_func:
        ._line  11
;## # C_SRC :           asm("   bset $b",s.bit1);
;#### ASM START
        bset 1,_s

;#### ASM END
        ._line  12
;## # C_SRC :   }
        rts

        .SECTION        bss_NO,DATA
        .glb    _s
_s:
        .blkb   1
```

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, 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. [*1]

```
asm("     op-code    R0, $@", variable name);
```

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

You can only specify one 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*2
        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
        ._line  10
;## # C_SRC :        asm("  mov.w  #1, $@", e_val);
;#### ASM START
        mov.w  #1, _e_val              ←――――――――――――――――   Reference to external variable
        ._line  11
;## # C_SRC :        asm("  mov.w  #2, $@", f_val);
        mov.w  #2, -2[FB]    ;  f_val  ←――――――――――――――――   Reference to auto variable
        ._line  12
;## # C_SRC :        asm("  mov.w  #3, $@", r_val);
        mov.w  #3, R0  ;  r_val        ←――――――――――――――――   Reference to register variable
        ._line  13
;## # C_SRC :        asm("  mov.w  #4, $@", s_val);
        mov.w  #4, ___S0_s_val         ←――――――――――――――――   Reference to static variable
;## # C_SRC :        asm("  mov.w  $@, $@", f_val,r_val);
        mov.w  -[FB], R0 ; f_val, r_val

;#### ASM END
```

Figure B.33 Example for Referencing Variables of Each Storage Class

*1 Whether it is arranged at which storage class should actually compile, and please check it.
*2 It does not restrict being assigned to a register, even if it specifies a register qualified.

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;

func()
{
    port.bit0 = 0x01;
    port.bit1 = 0x01;
}
```

Optimization ➡

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

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

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

Optimization ➡

Optimization is suppressed.

```
bset    00H,_port
bset    01H,_port
```

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)

(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     #0H,$$", i);       ⇐Format for checking register variables
```

Figure B.36 Example Coding of asm Function (2/2)

Note that, when you specify option -O[1-5], -OR, or -OS, 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.

Ex.:

```
void func ( int 1, int j )
{
        asm (“mov.w   2[FB],4[FB]”);// J=i;
}
```

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

(1)Do not destroy registers within the asm function. If registers are going to be destroyed, use push and pop instructions to save and restore the registers.

(2)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.37.

```
asm("   .SB    0");
asm("   LDC   #0H,  SB" );          ⇐SB changed
asm("   MOV.W    R0,_port[SB]");
              :
        (abbreviated)
              :
asm("   .SB    __SB__");            ⇐SB  returned to original state
asm("   LDC   #__SB__,SB");
```

Figure B.37 Restoring Modified Static Base (SB) register

Furthermore, pay careful attention to the functions that will be called while the SB register is modified and the interrupts that may occur during that time.

(3)Do not modified the FB register by the asm functions, because which use for the stack flame pointer.

c. Notes on Labels

The assembler source files generated by NC30 include internal labels in the format shown in Figure B.38. 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.38 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:

UNIX version	EUC (NCKIN, NCKOUT)
MS-Windows version	SJIS (NCKIN, NCKOUT)

- Environment variable specifying input code system .... NCKIN
- Environment variable specifying output code system .... NCKOUT

Figure B.39 is an example of setting the environment variables.

```
[UNIX]
This example sets the input to EUC codes and the output to Shift JIS codes.
% setenv NCKIN EUC
% setenv NCKOUT SJIS

[MS-Windows]
Include the following in your autoexec.bat file:
set NCKIN=SJIS
set NCKOUT=SJIS
```

Figure B.39 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.40 shows the format for including Japanese characters in character strings.

```
L" 漢字文字列 "
```

Figure B.40　Format of Kanji code Description in Character Strings

If you write Japanese using the format " 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.41 shows an example of a Japanese character string.

```
#include <stdlib.h>

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

         (remainder omitted)
             :
```

Figure B.41　Example of Japanese Character Strings Description

Figure B.42 is a memory map of the character string initialized in (1) in Figure B.41.

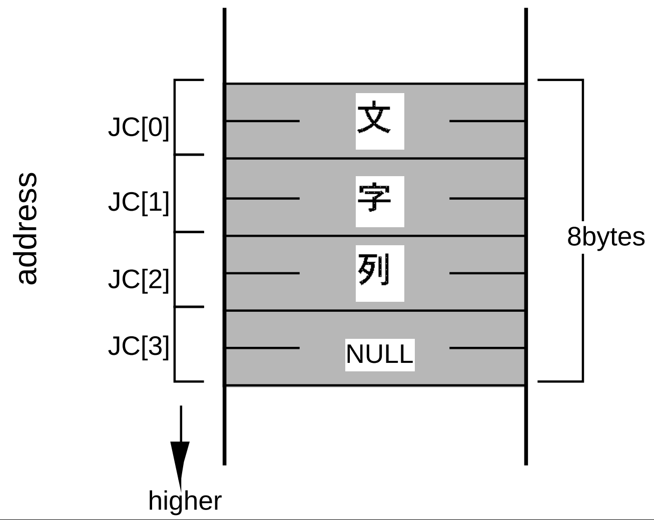


Figure B.42　Memory Location of wchar_t Type Character Strings

B.3.4 Using Japanese Characters as Character Constants

Figure B.43 shows the format for using Japanese characters as character constants.

```
L'漢'
```

Figure B.43 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.44 shows examples of how to write Japanese character constants.

```
#include <stdlib.h>

void near func()
{
        wchar_t JC[5];

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

          (remainder omitted)
              :
```

Figure B.44 Format of Kanji Character Constant Description

Figure B.45 is a memory map of the array to which the character constant in Figure B.44 has been assigned.

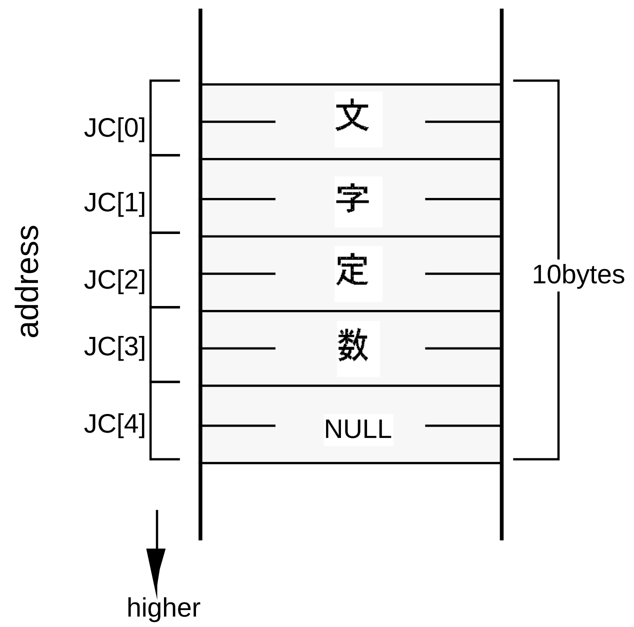


Figure B.45 Memory Location of wchar_t Type Character Constant Assigned Array

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.46 shows the format used to declare the default arguments of a function.

```
Storage class specifierΔType declaratorΔDeclarator([Dummy argument[=Default value or variable],...]);
```

Figure B.46 Format for declaring the default arguments of a function

Figure B.47 shows an example of declaration of a function, and Figure B.48 shows a result of compiling of sample program which shows at Figure B.47.

```
extern  int  func(int i=1, int j=2);
                :
        (abbreviated)
```

Figure B.47 Example for declaring the default arguments of a function

```
_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
      add.b    #02H,SP
      ._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.48   Compiling Result of smp1.c(smp1.a30)

A variable can be written for the argument of a function.

Figure B.49 shows an example where default arguments are specified with variables. Figure B.50 shows a compile result of the sample program shown in Figure B.49.

```
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.49   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.50 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.

- 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.51 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.51 Examples of Prototype Declaration

- 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.52 shows the format used to declare the inline storage class.

```
inlineΔtype specifierΔfunction;
```

Figure B.52 Declaration Format of inline Storage Class

Figure B.53 shows an example of declaration of a function.

```
int  s;
inline  int  func(int i);                ⇐Prototype declaration of function
{
        return ++i;
}
void main()                              ⇐Definition of body of function
{
      s=func(s);
}
```

Figure B.53 Example for Declaring inline Storage Class

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

;## M16C/60 C Compiler   OUTPUT
;## ccom30 Version X.XX.XX
;## COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
;## ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RE-
SERVED
;## Compile Start Time Thu April 10 18:40:11 1995,1996,1997,1998,1999,2000,2001,
2002,2003

;## COMMAND_LINE: ccom30  smp.i -o ./smp.a30 -dS


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

    .GLB  __SB__
    .SB   __SB__
    .FB   0

;## #       FUNCTION func

;## #       FUNCTION main
;## #       FRAME     AUTO (    i) size   2,     offset -4
;## #       FRAME     AUTO (    s)     size   2,     offset -2
;## #       ARG Size(0)  Auto Size(4)  Context Size(5)


    .SECTION    program,CODE,align
    ._file  'smp.c'
    .align
    ._line  7
;## # C_SRC :     {
    .glb    _main
_main:
    enter  #04H
    ._line  9
;## # C_SRC :            s = func(s);
    mov.w -2[FB],R0      ;  s
    ._line  2
;## # C_SRC :     {                          <---Inline storage class have codes
    mov.w R0,-4[FB]      ;  i                     embedded directly
    ._line  3
;## # C_SRC :            return i++;
    mov.w R0,R1
    add.w  #0001H,R0
    ._line  9
;## # C_SRC :            s = func(s);
    mov.w R1,-2[FB]      ;  s
    ._line  10
;## # C_SRC :     }
    exitd
E1:
    .END

;## Compile End Time Wed Nov 14 12:16:23 20xx
```

Figure B.54 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 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.

(2) Regarding the definition of an inline function

When specifying inline storage class for a function, be sure to define the body of the function in addition to declaring it. Make sure that this body definition is written in the same file as the function is written . The description in Figure B.55 is processed as an error in NC30.

```
inline void func(int i);

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

[Error Message]

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

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, your inline specification is ignored and all functions are handled as static functions. In this case, NC30 generates a warning. (See Figure B.57.)

```
int func(int i);

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

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

[Warning Message]

```
[Warning(ccom):smp.c,line 9] inline function is called as normal function before
,change to static function.
   ==⇒ {
```

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

(3) 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. (See Figure B.58.)

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

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

        f = &func;
}
```

[Error Message]

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

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

(4) 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. (See 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)

(5) 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
     :
     :
}
```

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 extended functions.

a. Using Memory Mapping Extended Functions

Table B.3 Memory Mapping Extended Functions

Extented function	Description
#pragma BIT	Declares that the external variable resides in an area where a 1-bit manipulate instruction can be used in 16-bit absolute addressing mode (i.e., a variable residing in addresses from 00000H to 01FFFH). Syntax : `#pragma BIT` *variable name* Example : `#pragma BIT bit_data`
#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 SBDATA	Declares that the data uses SB relative addressing. Syntax : `#pragma SBDATA` *variable name* Example : `#pragma SBDATA val`
#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`

*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.4 Extended Functions for Use with Target Devices

Extended function	Description
#pragma ADDRESS (#pragma EQU)	Specifies the absolute address of a variable. For near variables, this specifies the address within the bank. Syntax : `#pragma ADDRESS`Δ*variable-name*Δ*absolute-address* Example : `#pragma ADDRESS port0 2H` ❋ #pragma EQU can also be used for maintaining compatibility with C77.
#pragma INTCALL	Declares a function written in assembler called in a software interrupt (int instruction). Syntax : `#pragma INTCALL` Δ*INT-No.*Δ*function-name*(*register-name*) Example : `#pragma INTCALL 25 func(R0, R1)` Syntax : `#pragma INTCALL` *INT-No. function-name*() Example : `#pragma INTCALL 25 func()` ❋ Always be sure to declare the prototype of the function before entering this declaration.
#pragma INTERRUPT (#pragma INTF)	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`Δ`[/B|/E]`Δ*interrupt-handling-function-name* Example : `#pragma INTERRUPT int_func` Example : `#pragma INTERRUPT /B int_func` Example : `#pragma INTERRUPT /E int_func` ❋ #pragma INTF can also be used for maintaining compatibility with C77.
#pragma PARAMETER	Declares that, when calling an assembler function, the parameters are passed via specified registers. . Syntax : `#pragma PARAMETER`Δ*function_name* (*register_name*) 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`Δ*number function-name*() Example : `#pragma SPECIAL 30 func()`

c. Using MR30 Extended Functions

Table B.5 Extended Functions for MR30

Extended function	Description
#pragma ALMHANDLER	Declares the name of the MR30 alarm handler function Syntax : `#pragma ALMHANDLER` *function-name* Example : `#pragma ALMHANDLER alm_func`
#pragma CYCHANDLER	Declares the name of the MR30 cycle start handler function Syntax : `#pragma CYCHANDLER` *function-name* Example : `#pragma CYCHANDLER cyc_func`
#pragma INTHANDLER #pragma HANDLER	Declares the name of the MR30 interrupt handler function Syntax1 : `#pragma INTHANDLER` *function-name* Syntax2 : `#pragma HANDLER` *function-name* Example : `#pragma INTHANDLER int_func`
#pragma TASK	Declares the name of the MR30 task start function Syntax : `#pragma TASK` *task-start-function-name* Example : `#pragma TASK task1`

Supplement: The above extended function normally is generated by the configurator, so that the user need not be concerned with it.

d. The Other Extensions

Table B.6 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 R1,02H`
	`#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`

B.7.2 Using Memory Mapping Extended Functions

NC30 includes the following memory mapping extended functions.

#pragma BIT

1-bit Manipulate Instruction using Variable Declaration Function

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

Syntax : #pragma BIT∆*variable_name*

Description : The M16C/60 series 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;
                :
             (omitted)
                :
```

Figure B.62 Example Declaration of #pragma BIT

Note : 1-bit instructions 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 0000_{16} to address $01FFF_{16}$
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 ROM

Map to rom section

Function : Maps specified data (variable) to rom section

Syntax : #pragma ROMΔ*variable_name*

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

[1] Non-extern variables defined outside a function (Variables for which an area is secured)
[2] 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 an initialization expression.

Example :

[C language source program]

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

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

[Assembly language source program]

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

Figure B.63 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 SBDATA

SB Relative Addressing Using Variable Description Function

Function : Declares that the data uses SB relative addressing.

Syntax : #pragma SBDATAΔ*valuable-name*

Description : The M16C/60 series 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]

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;

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

Figure B.64 Example Use of #pragma SBDATA Declaration

*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 SECTIONΔ*section name*Δ*new section name*

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.

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:
```

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.

Note : In NC30WA V4.00 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 NC30WA V4.00 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.

#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. For example, the members of the structure in Figure B.66 are arranged in the order declared without any padding.

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

Member name	Type	Size	Mapped location (offset)
i	int	16 bits	0
c	char	8 bits	2
j	int	16 bits	3

Figure B.66 Example Mapping of Structure Members (1)

[1]Inhibiting packing

This NC30 extended function allows you to control the mapping of structure members. Figure B.67 is an example of mapping the members of the structure in Figure B.66 using #pragma STRUCT to inhibit packing.

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

Member name	Type	Size	Mapped location (offset)
i	int	16 bits	0
c	char	8 bits	2
j	int	16 bits	3
Padding	(char)	8 bits	–

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.

Description : [2]Arranging members

This NC30 extended function allows you to map the all odd-sized structure members first, followed by even-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	16 bits	0
j	int	16 bits	2
c	char	8 bits	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 EXT4MPTRΔ *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.

Example :

[C source program]

```
struct tag{
    int bitmap;
    char code;
}*pointer;

#pragma EXT4MPTR pointer

main()
{                                    ⇐Maps to pro1 section
    register int data;

    data=pointer->bitmap;
}
```

[Assembly language source program]

```
    mov.w    _pointer,A0
    mov.w    _pointer+2,A1
    bclr     3,A1              ⇐Change the bank
    bset     2,A1
    lde.w    [A1A0],-2[FB]
```

Figure B.70 Example Use of #pragma EXT4MPTR Declaration

Note : Before using this feature, check to see if the microcomputer and the system (hardware) support 4M-byte extension space mode.

B.7.3 Using Extended Functions for Target Devices

NC30 includes the following extended functions for target devices.

#pragma ADDRESS (#pragma EQU)

Specify absolute address of I/O variable

Function : Specifies the absolute address of a variable. For near variables, the specified address is within the bank.

Syntax : `#pragma ADDRESS`Δ*variable-name*;*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 .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. An error occurs if you include an initialization expression.
7. 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.

Example :

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

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

Figure B.71 #pragma ADDRESS Declaration

Note : For compatibility with C77 versions prior to V.2.10 before can accept files that include `#pragma EQU`. The absolute address using this format is written using the C conventions.

#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 INTCALLΔ[/C]Δ*INT-No.*Δ*assembler-function-name* (*register-name*, *register-name*, *...*)
(2)#pragma INTCALLΔ[/C]Δ*INT-No.*Δ*C-function-name* ( )

Description : This extended function declares the assembler function called by a software interrupt with the INT number.
When calling an assembler function, its parameters are passed via registers.
[/C] (NC308 ONLY)
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.

Rules : (1) 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 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 , R3R1 and A1A0
- far pointer types (24-bit registers) : R2R0, R3R1 and A1A0
- near pointer types (16-bit registers) : A0,A1,R0,R1,R2, and R3
- char 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.

(2) 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.
- **b.** 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.

Examples :

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

void main()
{
     int     i;
     long     l;
     i = 0x7FFD;
     l = 0x007F;

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

Figure B.72 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()
{
     int     i, j;
     i = 0x7FFD;
     j = 0x007F;

     c_func( i, j );                        ⇐Calling the C function

}
```

Figure B.73 Example of #pragma INTCALL Declaration(C language functuion) (2)

#pragma INTERRUPT (#pragma INTF)

Declare interrupt function

Function : Declares an interrupt handler

Syntax : #pragma INTERRUPT∆[/B|/E|/F]∆*interrupt-handler-name*

Description : 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.

You may specify either /B or /E of /F 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.

/E : Multiple interrupts are enabled immediately after entering the interrupt. This improves interrupt response.

/F (NC308 ONLY)
:Return to th calling function by the FREIT instruction in exit processing.

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.

Example :

```
#pragma INTERRUPT i_func


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

Figure B.74 Example of #pragma INTERRUPT Declaration

Note : For compatibility with C77 versions prior to V.2.10 before can accept files that include `#pragma INTF`.

#pragma PARAMETER

Declare assembler function that passed arguments via register

Function : Declares an assembler function that passes parameters via registers

Syntax : `#pragma PARAMETER`Δ[/C]Δ*assembler-function-name* (*register-name*, *register-name*,…)

Description :

This extended function declares that, when calling an assembler function, its parameters are passed via registers.

- float types, long types (32-bit registers) : R2R0 ,R3R1 and A1A0
- far pointer types (24-bit registers) : R2R0, R3R1 and A1A0
- near pointer types (16-bit registers) : A0,A1,R0,R1,R2, and R3
- char types (8-bit registers) : R0L, R0H, R1L, and R1H

* Register names are NOT case-sensitive.

[/C] (NC308 ONLY)
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.

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

c. The assembler functions shown below cannot be declared:

- Functions returning structure or union type

3. An error occurs, when you write the function entity specified in `#pragma PARAMETER` in C language.

Example :

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

void main()
{
     int     i, j;
     i = 0x7FFD;
     j = 0x007F;

     asm_func( i, j );                    ⇐Calling the assembler function

}
```

Figure B.75 Example of #pragma PARAMETER Declaration

#pragma SPECIAL

Declare a special page subroutine call function

Function : Declares a special page subroutine call (JSRS instruction) function

Syntax : #pragma SPECIAL∆[/C]∆*number*∆*function-name*()

Description : 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.

[/C] (NC308 ONLY)
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.

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

Example :

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

void main()
{
     int     i, j;
     i = 0x7FFD;
     j = 0x007F;

     func( i, j );                    ⇐special page subroutine call

}
```

Figure B.76 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 Using MR30 Extended Functions

NC30 has the following extended functions which support the real-time operating system MR30.

#pragma ALMHANDLER

Alarm handler declaration

Function : Declares an MR30 alarm handler

Syntax : #pragma ALMHANDLER∆*alarm-handler-name*

Description : By using the above format to declare an alarm handler (a function) written in C, NC30 generates the code for the alarm handler to be used at the entry and exit points of the function.

- The alarm handler is called from the system clock interrupt by the JSR instruction and returns by the RTS or EXITD instruction.

Rules :
1. You canNOT write alarm handlers that take parameters.
2. The return value from the alarm handler must be type void in the declaration.
3. Only the function definition put after `#pragma ALMHANDLER` are valid.
4. No processing occurs if you specify other than a function name.
5. No error occurs if you duplicate `#pragma ALMHANDLER` declarations.
6. A compile error occurs if you use any function specified in one of the following declarations in `#pragma ALMHANDLER`:
   - #pragma INTERRUPT
   - #pragma INTHANDLER
   - #pragma HANDLER
   - #pragma CYCHANDLER
   - #pragma TASK

Example :

```
#include   <mr30.h>
#include   "id.h"

#pragma ALMHANDLER      alm

void alm(void)         ⇐Be sure to declare as type void.
{
          :
      (omitted)
          :
}
```

Figure B.77 Example of #pragma ALMHANDLER Declaration

#pragma CYCHANDLER

Cyclic handler declaration

Function : Declares an MR30 cyclic handler

Syntax : #pragma CYCHANDLER∆*cyclic-handler-name*

Description : By using the above format to declare a cyclic handler (a function) written in C, NC30 generates the code for the cyclic handler to be used at the entry and exit points of the function.

- The cyclic handler is called from the system clock interrupt by the JSR instruction and returns by the RTS or EXITD instruction.

Rules :

1. You canNOT write cyclic handlers that take parameters.
2. The return value from the cyclic handler must be type void in the declaration.
3. Only the function definition put after `#pragma CYCHANDLER` are valid.
4. No processing occurs if you specify other than a function name.
5. No error occurs if you duplicate `#pragma CYCHANDLER` declarations.
6. A compile error occurs if you use any function specified in one of the following declarations in `#pragma CYCHANDLER`:
   - #pragma INTERRUPT
   - #pragma INTHANDLER
   - #pragma HANDLER
   - #pragma ALMHANDLER
   - #pragma TASK

Example :

```
#include   <mr30.h>
#include   "id.h"

#pragma CYCHANDLER      cyc

void cyc(void)        ⇐Be sure to declare as type void.
{
          :
      (omitted)
          :
}
```

Figure B.78 Example of #pragma CYCHANDLER Declaration

#pragma INTHANDLER (#pragma HANDLER)

Interrupt handler declaration

Function : Declares an MR30 OS-dependent interrupt handler

Syntax : [1] #pragma INTHANDLERΔi*interrupt-handler-name*
[2] #pragma HANDLERΔ*interrupt-handler-name*

Description : By using the above format to declare an interrupt handler (a function) written in C, NC30 generates the code for the handling shown below to be used at the entry and exit points of the function :

1. At the entry point : Push (i.e., save) the registers onto the current stack.
2. At the exit point : Returns from the interrupt with the ret_int system call. Also returns from the interrupt by the ret_int system call when returning at a return statement partway through the function.

To declare an MR30 OS-independent interrupt handler, use `#pragma INTERRUPT`.

Rules :

1. You canNOT write interrupt handlers that take parameters.
2. The return value from the interrupt handler must be type void in the declaration.
3. Do NOT use the ret_int system calls from C.
4. Only the function definition put after `#pragma INTHANDLER` are valid.
5. No processing occurs if you specify other than a function name.
6. No error occurs if you duplicate `#pragma INTHANDLER` declarations.
7. A compile error occurs if you use any function specified in one of the following declarations in `#pragma INTHANDLER`:
   - #pragma INTERRUPT
   - #pragma HANDLER
   - #pragma ALMHANDLER
   - #pragma CYCHANDLER
   - #pragma TASK

Example :

```
#include   <mr30.h>
#include   "id.h"

#pragma INTHANDLER      hand

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

Figure B.79 Example of #pragma INTHANDLER Declaration

#pragma TASK

Task start function declaration

Function : Declares an MR30 task start function

Syntax : #pragma TASKΔ*task-start-function-name*

Description : By using the above format to declare a task start function written in C, NC30 generates the code for processing for the task shown below to be used at the exit points of the function.

- At the exit point : Ends by the ext_tsk system call. Also returns using the ext_tsk system call even when returning at a return statement part way through function.

Rules :
1. You need not put the ext_tsk system call to return from the task.
2. The return value from the task must be type void in the declaration.
3. Only the function definition put after `#pragma TASK` are valid.
4. No processing occurs if you specify other than a function name.
5. No error occurs if you duplicate `#pragma TASK` declarations.
6. A compile error occurs if you use any function specified in one of the following declarations in `#pragma TASK`:
   - #pragma INTERRUPT
   - #pragma INTHANDLER
   - #pragma HANDLER
   - #pragma ALMHANDLER
   - #pragma CYCHANDLER

Example :

```
#include   <mr30.h>
#include   "id.h"

#pragma TASK      main
#pragma TASK      tsk1

void main(void)             ⇐Be sure to declare as type void.
{
          :
      (omitted)
          :
       sta_tsk(ID_idle);
       sta_tsk(ID_tsk1);
       /* ext_tsk(); */      ⇐You need not use ext_tsk.
}

void tsk1()
          :
    (remainder omitted)
```

Figure B.80   Example of #pragma TASK Declaration

B.7.5 The Other Extensions

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

#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

Rules : Writing `#pragma ASM`, be sure to use it in combination with `#pragma ENDASM`. NC30 suspends processing if no `#pragma ENDASM` is found the corresponding `#pragma ASM`.

1. In assembly language description, do not write statements which will cause the register contents to be destroyed. 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.

Example :

```
void func()
{
    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.81 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 JSRAΔ*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.82 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*

Description : 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.83 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()
{
    int     i, j;

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

#pragma PAGE

    i++;

}
```

Figure B.84 Example of #pragma PAGE

#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 and _Bool types: R0L float types : R2R0 int and short types: R0 double types : R3R2R1R0 long types: R2R0 long-long type R3R1R2R0.

(5) f 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.wR2,R0 ; The return-value is set to R2R0 register
    .endm
#pragma ENDASM
long l;
void test func(void)
{
    l = mul(2,3);
}
```

Figure B.85   Example of #pragma __AMMACRO

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];

func()
{
        l = rmpa_b(1,19,a,b);    /* asm Macro Function(rmpa command) */
}
```

Figure B.85   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.

DADD

Function : Returns the result of decimal addition on val1 plus val2.

Syntax :
```
#include <asmmacro.h>
unsigned char     dadd_b(unsigned char val1, unsigned char val2);
                                          /* When calculated in 8 bits */
unsigned  int     dadd_w(unsigned int  val1,unsigned  int  val2);
                                          /* When calculated in 16 bits */
```

DADC

Function : Returns the result of decimal addition with carry on val1 plus val2.

Syntax :
```
#include <asmmacro.h>
unsigned char     dadc_b(unsigned char val1, unsigned char val2);
                                          /* When calculated in 8 bits */
unsigned  int     dadc_w(unsigned int  val1,unsigned   int  val2);
                                          /* When calculated in 16bits */
```

DSUB

Function : Returns the result of decimal subtraction on val1 minus val2.

Syntax :
```
#include <asmmacro.h>
unsigned char     dsub_b(unsigned char val1, unsigned char val2);
                                          /* When calculated in 8 bits*/
unsigned int      dsub_w(unsigned int  val1,unsigned  int  val2);
                                          /* When calculated in 16 bits */
```

DSBB

Function : Returns the result of decimal subtraction with borrow on val1 minus val2.

Syntax :
```
#include <asmmacro.h>
unsigned char     dsbb_b(unsigned char val1, unsigned char val2);
                                          /* When calculated in 8 bits */
unsigned  int     dsbb_w(unsigned int  val1,unsigned  int  val2);
                                          /* When calculated in 16 bits */
```

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>
signed long rmpa_b(signed long init,signed  int count, signed char *p1,signed  char*p2);
                                          /* When calculated in 8 bits */
signed long rmpa_w(signed long init,signed  int count,signed  int *p1, signed int *p2);
                                          /* When calculated in 16 bits*/
```

DIV

Function: devide val1 by val2

Syntax :
```
#include <asmmacro.h>
signed char       div_b(signed int val1, signed char val2);
                                          /* calculated in 8 bits with signed*/
signed int        div_w(signed int  val1,signed  int  val2);
                                          /* calculated in 16 bits with signed*/
unsigned char     divu_b(unsigned int val1,unsigned char val2);
                                          /*calculated in 8 bits with unsigned */
unsigned int      divu_w(unsigned long val1,unsigned int val2);
                                          /*calculated in 16 bits with unsigned */
signed char       divx_b(signed int val1,signed char val2);
                                          /* calculated in 8 bits with signed*/
signed int        divx_w(signed long val1,signed int val2);
                                          /* calculated in 16 bits with signed*/
```

MOD

Function : devide val1 by val2 and get mod.

Syntax : #include <asmmacro.h>

```
signed char      mod_b(signed int val1,signed char val2);
                                                    /* When calculated in 8 bits */
signed int       mod_b(signed long  val1, signed int  val2);
                                                    /*  calculated in 16 bits */
unsigned char    modu_b(unsigned int val1,unsigned char val2);
                                                    /*calculated in 8 bits */
unsigned int     modu_w(unsigned long val1,unsigned int val2);
                                                    /*calculated in 16 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 address-decrementing direction. There is no return value.

Syntax :

```
void      smovb_b(unsigned char *p1, unsigned char *p2, unsigned int count);
                                                    /*calculated in 8 bits */
void      smovb_w(unsigned int *p1,unsigned  int *p2, unsigned int count);
                                                    /* 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 :

```
void      smovf_b(unsigned char *p1, unsigned char *p2, unsigned int count);
                                                    /*calculated in 8 bits */
void      smovf_w(unsigned int  *p1,unsigned  int  *p2, unsigned int count);
                                                    /*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 :
```
void      sstr_b(unsigned char val,unsigned  char *p, unsigned int count);
                                                  /*calculated in 8 bits */
void      sstr_w(unsigned int val,unsigned  int *p, unsigned int count);
                                                  /*calculated in 16 bits*/
```

MOVdir

Function : transfer to val2 from val1 by nibble

Syntax :
```
#include <asmmacro.h>
unsigned char    movll(unsigned char val1,unsigned  char val2);
                                   /* to low of val2 from high of val1 */
unsigned char    movlh(unsigned char val1,unsigned  char val2);
                                   /* to high of val2 from low of val1*/
unsigned char    movhl(unsigned char val1, unsigned char val2);
                                   /* to low of val2 from high of val1 */
unsigned char    movhh(unsigned char val1,unsigned char val2);
                                   /* to high of val2 from high of val1 */
```

ROLC

Function : The value of val is returned after rotating it left by 1 bit including the C flag.

Syntax :
```
#include <asmmacro.h>
unsigned char    rolc_b(unsigned char val1);      /* When calculated in 8 bits */
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>
unsigned char    rorc_b(unsigned char val);    /* When calculated in 8 bits */
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 <asmmcaro.h>
unsigned char     rot_b(signed char count, unsigned char val);
                                            /* When calculated in 8 bits */
unsigned int      rot_w(signed char count, unsigned int val);
                                            /* When calculated in 16 bits */
```

SHA

Function : The value of val is returned after arithmetically shifting it as many times as indicated by count.

Syntax :

```
#include <asmmacro.h>
unsigned char     sha_b(signed char count, unsigned char val);
                                            /* When calculated in 8 bits */
unsigned int      sha_w(signed char count, unsigned int val);
                                            /* When calculated in 16 bits */
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>
unsigned char     shl_b(signed char count, unsigned char val);
                                            /* When calculated in 8 bits */
unsigned int      shl_w(signed char count, unsigned int val);
                                            /* When calculated in 16 bits */
unsigned long     shl_l(signed char count, unsigned long val);
                                            /* When calculated in 24 bits */
```

ABS

Function : absolute

Syntax :

```
#include <asmmacro.h>
signed char       abs_b(signed char val);     /* When calculated in 8 bits */
signed int        abs_w(signed  int val);     /* When calculated in 16 bits */
```

NEG

Function : negate

Syntax :

```
#include <asmmacro.h>
signed char    neg_b(signed char val);    /* When calculated in 8 bits */
signed int     neg_w(signed  int val);    /* When calculated in 16 bits */
```

NOT

Function : not

Syntax :

```
#include <asmmacro.h>
signed char    not_b(signed char val);    /* When calculated in 8 bits */
signed int     not_w(signed  int val);    /* When calculated in 16 bits */
```

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

NC30 is a cross C compiler targeting the M16C/60,20 series. 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/60,20 series 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

Table C.1 shows the standard EWS environment assumed when testing performance. TableC.2 shows the standard PC environment.

TableC.1 Standard EWS Environment

Item	Type of EWS	UNIX Version
EWS environment	SPARCstation	SunOS V.4.1.3 JLE1.1.3
		Nihongo Solaris 2.5
	HP 9000/700 Series	HP-UX V.10.20
Available swap area	100MB min.	

TableC.2 Standard PC Environment

Item	Type of PC	OS Version
PC environment	IBM PC/AT or compatible	Windows ME Windows 2000
Type of CPU	Intel Pentium II	
Memory	128MB min.	

TableC.3 Standard Linux Environment

Item	Type of PC	OS Version
PC environment	IBM PC/AT or compatible	Turbo Linux 7.0
Type of CPU	Intel Pentium II	
Memory	128MB min.	

b. C Source File Coding Specifications

Table C.4 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.4 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

Table C.5 to C.5 lists the NC30 specifications. Note that estimates are provided for items for which actual measurements could not be achieved.

Table C.5 NC30 Specifications

Item	Specification
Maximum number of files that can be specified in NC30	Depends on amount of available memory
Maximum length of filename	Depends on operating system
Maximum number of macros that can be specified in nc30 command line option -D	Depends on amount of available memory
Maximum number of directories that can be specified in nc30 command line option -I	8max
Maximum number of parameters that can be specified in nc30 command line option -as30	Depends on amount of available memory
Maximum number of parameters that can be specified in nc30 command line option -ln30	Depends on amount of available memory
Maximum nesting levels of compound statements,iteration control structures, and selection control structures	Depends on amount of available memory
Maximum nesting levels in conditional compiling	Depends on amount of available memory
Number of pointers modifying declared basic types, arrays, and function declarators	Depends on amount of available memory
Number of function definitions	Depends on amount of available memory
Number of identifiers with block scope in one block	Depends on amount of available memory
Maximum number of macro identifiers that can be simultaneously defined in one source file	Depends on amount of available memory
Maximum number of macro name replacements	Depends on amount of available memory
Number of logical source lines in input program	Depends on amount of available memory
Maximum number of levels of nesting #include files	40max
Maximum number of case names in one switch statement (with no nesting of switch statement)	Depends on amount of available memory
Total number of operators and operands that can be defined in #if and #elif	Depends on amount of available memory
Size of stack frame that can be secured per function(in bytes)	255 max
Number of variables that can be defined in #pragma ADDRESS	Depends on amount of available memory
Maximum number of levels of nesting parentheses	Depends on amount of available memory
Number of initial values that can be defined when defining variables with initialization expressions	Depends on amount of available memory
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	Depends on amount of available memory
Maximum number of valid characters per external identifier	Depends on amount of available memory
Maximum number of external identifiers per source file	Depends on amount of available memory
Maximum number of identifiers with block scope per block	Depends on amount of available memory
Maximum number of macros per source file	Depends on amount of available memory
Maximum number of parameters per function call and per function	Depends on amount of available memory
Maximum number of parameters or macro call parameters per macro	31max
Maximum number of characters in character string literals after concatenation	Depends on amount of available memory
Maximum size (in bytes) of object	Depends on amount of available memory
Maximum number of members per structure/union	Depends on amount of available memory
Maximum number of enumerator constants per enumerator	Depends on amount of available memory
Maximum number of levels of nesting of structures or unions per struct declaration list	Depends on amount of available memory
Maximum number of characters per character string	Depends on operating system
Maximum number of lines per file	Depends on amount of available memory

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

a. Key Words

NC30 interprets the followings as key words.

Table C.6 Key Words List

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

In the entry version, the keywords listed below are not handled as keywords:

near far inline asm

When using these keywords, add the underscore "_" before the first character of each keyword used.

_near _far _inline _asm

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 31 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. Table C.7 shows the format of each base (decimal, octal, and hexadecimal).

Table C.7 Specifying Integer Constants

Base	Notation	Structure	Example
Decimal	None	0123456789	15
Octal	Start with 0 (zero)	01234567	017
Hexadecimal	Start with 0X or 0x	0123456789ABCDEF 0123456789abcdef	0XF or 0xf

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

- Octal and hexadecimal: signed int ⇒unsigned int ⇒signed long ⇒unsigned long ⇒signed long long ⇒unsigned long long
- Decimal : signed int ⇒signed long ⇒unsigned long ⇒signed long long ⇒unsigned 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: signed long ⇒unsigned long ⇒signed long long ⇒unsigned long long
- Decimal : signed long long ⇒unsigned long long

[3]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: 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 \.

Table C.8   Extended Notation List

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

d. Character Literals

Character literals are written in double quote marks, as in "character string". The extended notation shown in Table C.8 for character constants can also be used for character literals.

e. Operators

NC30 can interpret the operators shown in Table C.9.

Table C.9 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
- ewstrict
- near
- far

c. Data Type and Size

Table C.10 shows the size corresponding to data type.

Table C.10 Data Type and Bit Size

Type	Existence of sign	Bit size	Range of values
_Bool	No	8	0, 1
char	No	8	0↔255
unsigned char			
signed char	Yes	8	-128↔127
int	Yes	16	-32768↔32767
short			
signed int			
signed short			
unsigned int	No	16	0↔65535
unsigned short			
long	Yes	32	-2147483648↔2147483647
signed long			
unsigned long	No	32	0↔4294967295
long long	Yes	64	-9223372036854775808↔ 9223372036854775807
signed long long			
unsigned long long	No	64	18446744073709551615
float	Yes	32	1.17549435e-38F↔3.40282347e+38F
double	Yes	64	2.2250738585072014e-308↔ 1.7976931348623157e+308
long double			
near pointer	No	16	0↔0xFFFF
far pointer	No	32	0↔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

Tables C.11 and Table C.12 show the relationship between types of expressions and their elements.

Table C.11 Types of Expressions and Their Elements (1/2)

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

Table C.12   Types of Expressions and Their Elements (2/2)

Type of expression	Elements of 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
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 specifierΔtype declaratorΔdeclaration specifierΔinitialization_expression;

Figure C.1 Declaration Format of Variable

(1)Storage-class Specifiers

NC30 supports the following storage-class specifiers.

- extern
- auto
- typedef
- static
- register

(2)Type Declarator

NC30 supports the type declarators.

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

(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 specifierΔtype declaratorΔdeclaration specifierΔmain program

- function declaration (prototype declaration)

storage-class specifierΔtype declaratorΔdeclaration 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	● char	● int	● short	● long
● long long	● float	● double	● unsigned	● signed
● struct	● union	● enum		

(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 expression$_{opt}$]
		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 Declarator$_{opt}$ 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
- Compound Statement
- Expression / Null Statement
- Selection Statement
- Iteration Statement
- Jump Statement
- Assembly Language 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 declarations$_{opt}$list of statements$_{opt}$ }

* 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

Table C.13 lists the preprocess commands available in NC30.

Table C.13 List of Preprocess Commands

Command	Function
#define	Defines macros.
#undef	Undefines macros.
#include	Takes in the specified file.
#error	Outputs messages to the standard output device and terminates processing.
#line	Specifies file's line numbers.
#assert	Outputs a warning when a constant expression is false.
#pragma	Instructs processing for NC30's extended function.
#if	Performs conditional compilation.
#ifdef	Performs conditional compilation.
#ifndef	Performs conditional compilation.
#elif	Performs conditional compilation.
#else	Performs conditional compilation.
#endif	Performs conditional compilation.

C.3.2 Preprocess Commands Reference

The NC30 preprocess commands are described in more detail below. They are listed in the order shown in Table C.13.

#define

[Function] Defines macros.

[Format] [1]#define∆indentifier∆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
NC30
```

● 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)
```

#define

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

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

#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)
```

#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.

#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.

#line

[Function] Changes the line number in the file.

[Format] #line∆integer∆"file name"

[Description]
- Specify the line number in the file and the filename.
- You can change the name of the source file and the line No.

#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
```

#pragma

[Function] Instructs the system to process NC30's extended functions.

[Format]
1. #pragma ROMΔvariable name
2. #pragma SBDATAΔvariable name
3. #pragma SECTIONΔpredetermined section nameΔaltered section name
4. #pragma STRUCTΔtag name of structureΔunpack
4. #pragma STRUCTΔtag name of structureΔarrange
5. #pragma EXT4MPTRΔname of pointer
6. #pragma ADDRESSΔvariable nameΔabsolute address
6. #pragma EQUΔvariable name = absolute address
7. #pragma INTCALLΔint No.Δassembler function name(register name, register name, ..)
7. #pragma INTCALLΔint No.ΔC language function name()
8. #pragma INTERRUPTΔ[/B | /E]Δinterrupt handling function name
8. #pragma INTFΔinterrupt handling function name
9. #pragma PARAMETERΔassembler function name(register name, register name, ..)
10. #pragma SPECIALΔspecial No.Δfunction name
11. #pragma ALMHANDLERΔalarm handler function name
12. #pragma CYCHANDLERΔcyclic handler function name
13. #pragma INTHANDLERΔinterrupt handler function name
13. #pragma HANDLERΔinterrupt handler function name
14. #pragma TASKΔtask start function name
15. #pragma ASM
15. #pragma ENDASM
16. #pragma JSRAΔfunction name
17. #pragma JARWΔfunction name
18. #pragma PAGE

[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 declare pointer for access 4M-byte ROM area
6. Facility to specify absolute addresses for input/output variables
7. Facility to declare functions using software interrupts
8. Facility to write interrupt functions
9. Facility to declare assembler functions passed via register
10. Facility to declare special page subroutine call functions
11. Facility to describe alarm handler functions
12. Facility to describe cyclic handler functions
13. Facility to describe interrupt handler functions
14. Facility to describe taskstart functions
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

- You can only specify the above 17 processing functions with #pragma. If you specify a character string or identifier other than the above after #pragma, it will be ignored.
- Always use uppercase to specify the process (SECTION, INTERRUPT, etc.).
- 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).

#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).
- You cannot use the sizeof operator, cast operator, or variables in a constant expression.

#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).
- You cannot use the sizeof operator, cast operator, or variables in a constant expression.

#ifndef - #elif - #else - #endif

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

[Format]

```
#ifndefΔidentifier
        :
#elifΔconstant expression
        :
#else
        :
#endif
```

[Description]

- If an identifier isn't defined, #ifndef processes the program that follows.You can also describe the followings.

```
#ifΔ !definedΔ identifier
#ifΔ !definedΔ (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.

C.3.3 Predefined Macros

The following macros are predefined in NC30:

- M16C
- NC30

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           portB = 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

Table D.1 shows the number of bytes used by integral type data.

D.1.1 Integral Type

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↔255
signed char	Yes	8	-128↔127
int short signed int signed short	Yes	16	-32768↔32767
unsigned int unsigned short	No	16	0↔65535
long signed long	Yes	32	-2147483648↔2147483647
unsigned long	No	32	0↔4294967295
long long signed long long	Yes	64	-9223372036854775808↔ 9223372036854775807
unsigned long long	No	64	18446744073709551615
float	Yes	32	1.17549435e-38F↔3.40282347e+38F
double long double	Yes	64	2.2250738585072014e-308↔ 1.7976931348623157e+308
near pointer	No	16	0↔0xFFFF
far pointer	No	32	0↔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.

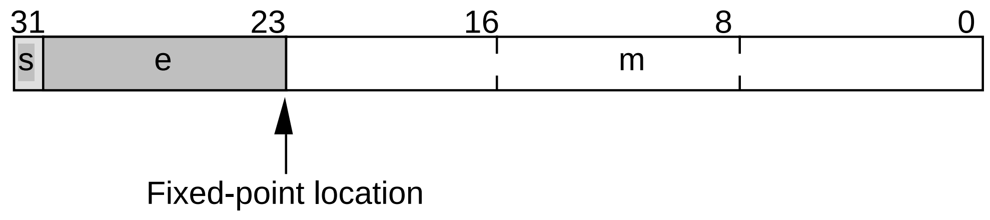
Table D.2　Data Size of Floating Type

Type	Existence of sign	Bit Size	Range of values
float	Yes	32	1.17549435e-38F↔3.40282347e+38F
double	Yes	64	2.2250738585072014e-308↔ 1.7976931348623157e+308
long double			

NC30's floating-point format conforms to the format of IEEE (Institute of Electrical and Electronics Engineers) standards. The following shows the single precision and double precision floating-point formats.

(1)Single-precision floating point data format

Figure D.1 shows the format for binary floating point (float) data.

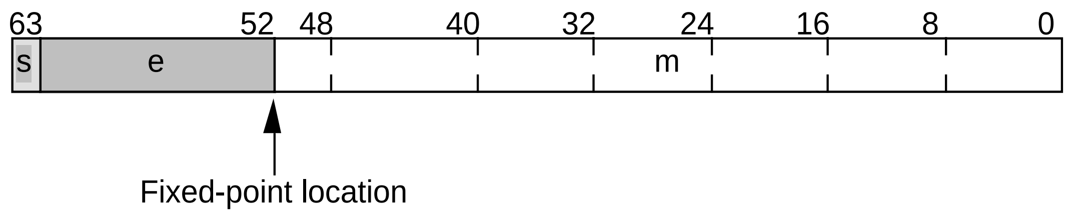


s:fixed-point part sign(1bit)
e:characteristic part(8bits)
m:fixed-point part(23bits)

Figure D.1　Single-precision floating point data format

(2)Double-precision floating point data format

Figure D.2 shows the format for binary floating point (double and long double) data.



s:fixed-point part sign(1bit)
e:characteristic part(11bits)
m:fixed-point part(52bits)

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	0-0xFFFF
far pointers	None	32	0-0xFFFFFFFF

Note that only the least significant 24 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.

[Example]

```
char  c[5] = {0, 1, 2, 3, 4};
```

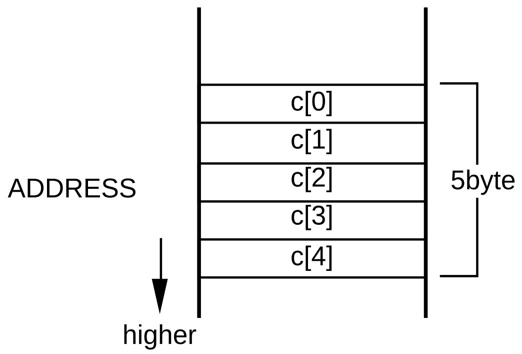


Figure D.3 Example of Placement of Array

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;
```

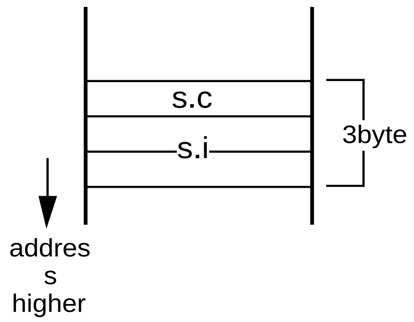


Figure D.4 Example of Placement of Structure(1/2)

Normally, there is no word alignment with structures. The members of structures are mapped contiguously. To use word alignment, use the #pragma STRUCT extended function. #pragma STRUCT adds a byte of padding if the total size of the members is odd. Figure D.5 is an example of mapping.

(Example)

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

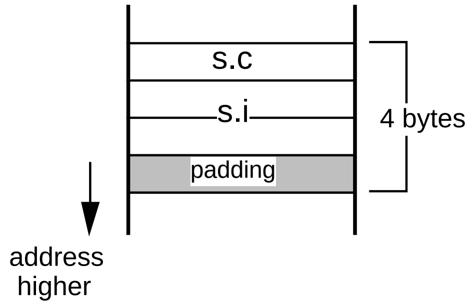


Figure D.5   Example of Placement of Structure(2/2)

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;
```

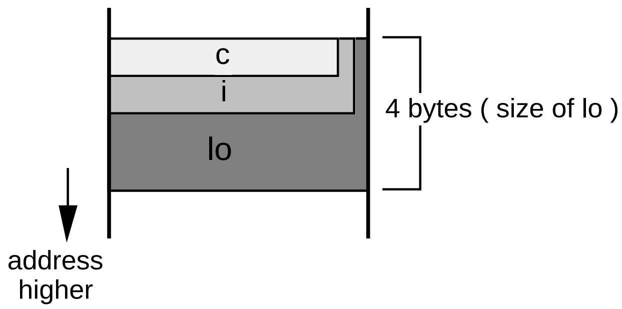


Figure D.6   Example of Placement of Union

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;
```

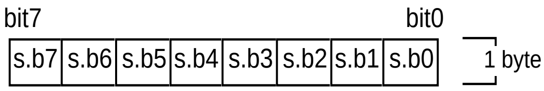


Figure D.7 Example of Placement of Bitfield(1/2)

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;
```

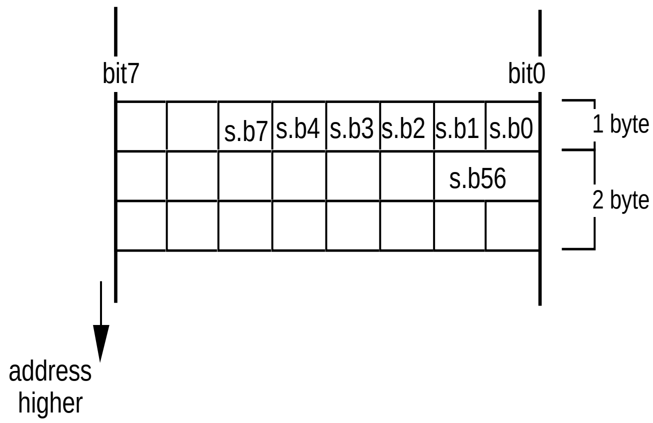


Figure D.8 Example of Placement of Bitfield(2/2)

Note :

- If no sign is specified, the default bitfield member type is unsigned.
- 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.

```
func()
{
      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 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 D.6 are matched.

Table D.5 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

Table D.6 Rules on Argument Transfer via Register(NC308)

Argument	First Argument	Registers Used
First argument	char type, _Bool type	R1L register
	int type near pointer type	R1 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 D.8 summarize the methods used to pass arguments.

Table D.7 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 register	Stack
Other types	Stack	Stack	Stack

Table D.8 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

*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.

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.

```
int func_proto( int , int , int);          ←[1]

                                                 [2]
int func_proto(int i, int j, int k)
{
        return i + j + k;
}
                                                 [3]
int func_no_proto( i, j, k)
int i;
int j;
int k;
{
        return i + j + k;
}
                                                 [4]
void
main(void)
{
        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.

*1. However, function names are not output for the functions that are specified by #pragma INTCALL.

```
;## #   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(4)    Context Size(5)

        .section        program
        ._file  'proto.c'
        ._line  4
;## # C_SRC :   {
        .glb    $func_proto                 ←[10]
$func_proto:
        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
```

[7]This passes the third argument k via stack.
[8]This passes the first argument i via register.
[9]This passes the second argument j via register.
[10]This is the start address of function func_proto.

Figure D.11 Compile Result of Sample Program (sample.c) (1/3)

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)

        ._line  11
;## # C_SRC :   {
        .glb    _func_no_proto←[12]
_func_no_proto:
        enter   #00H
        ._line  12
;## # 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
```

[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) (2/3)

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)

        ._line  16
;## # C_SRC :   {
        .glb    _main
_main:
        enter   #02H
        ._line  18
;## # C_SRC :           sum = func_proto(1,2,3);    [11]
        push.w  #0003H
        mov.w   #0002H,R2
        mov.w   #0001H,R1
        jsr     $func_proto
        add.b   #02H,SP
        mov.w   R0,-2[FB]       ;  sum
        ._line  19
;## # C_SRC :           sum = func_no_proto(1,2,3);  [12]
        push.w  #0003H
        push.w  #0002H
        push.w  #0001H
        jsr     _func_no_proto
        add.b   #06H,SP
        mov.w   R0,-2[FB]       ;  sum
        ._line  20
;## # C_SRC :   }
        exitd
        .END
```

Figure D.13 Compile Result of Sample Program (sample.c) (3/3)

In Figure D.13, part[11]calls func_proto and part[12]calls func_no_proto.

D.3.4 Interface between Functions

Figures 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 = 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
;## #   ARC Size(0)        Auto Size(6)        Context Size(5);

        .section        program
        ._file  'proto2.c'
        ._line  5
        .glb    _main
_main:                                    ←[1]
        enter   #06H                      ←[2]
        ._line  6
;## # C_SRC :         int i = 0x1234;
        mov.w   #1234H,-2[FB]   ;  i
        ._line  7
;## # C_SRC :         int j = 0x5678;
       mov.w    #5678H,-4[FB]   ;  j
        ._line  8
;## # C_SRC :         int k = 0x9abc;
        mov.w   #9abcH,-6[FB]   ;  k
        ._line  9
;## # C_SRC :         k = func(i,j,k);
        push.w  -6[FB]  ;  k              ←[3]
        mov.w   -4[FB],R2  ;  j           ←[4]
        mov.w   -2[FB],R1       ;  i      ←[5]
        jsr     $func                     ←[6]
        add.b   #02H,SP                   ←[10]
        mov.w   R0,-2[FB]       ;  k      ←[11]
        ._line  10
        exit
```

Figure D.15 Assembly language sample program (1/2)

```
;## #   FUNCTION func
;## #   FRAME   AUTO    (        x)      size   2,       offset -2
;## #   FRAME   AUTO    (      sum)      size   2,       offset -2
;## #   FRAME   ARG     (        y)      size   2,       offset 5
;## #   FRAME   ARG     (        z)      size   2,       offset 8
;## #   REGISTER ARG    (        x)      size   2,       REGISTER R0
;## #   ARG Size(4)     Auto Size(2)     Context Size(8)

        ._line  13
;## # C_SRC :      {
        .glb    $func
$func:
        enter   #02H                           ←[7]
        mov.w   R0,-2[FB]       ;  x  x
        ._line  16
;## # C_SRC :          sum = x + y + z;
        mov.w   -2[FB],R0       ;  x
        add.w   8[FB],R0        ;  y
        add.w   10[FB],R0       ;  z
        mov.w   R0,-2[FB]       ;  sum
        ._line  17
;## # C_SRC :         return sum;
        mov.w   -2[FB],R0       ;  sum         ←[8]
        exitd                                  ←[9]
        .END
```

Figure D.16 Assembly language sample program (2/2)

Figures 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.

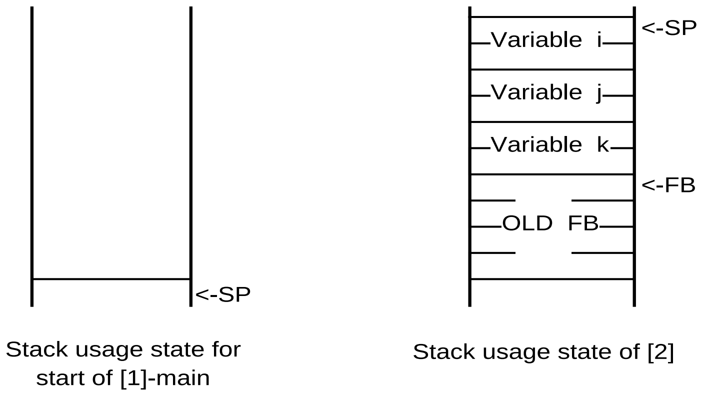


Figure D.17   Entry processing of function main

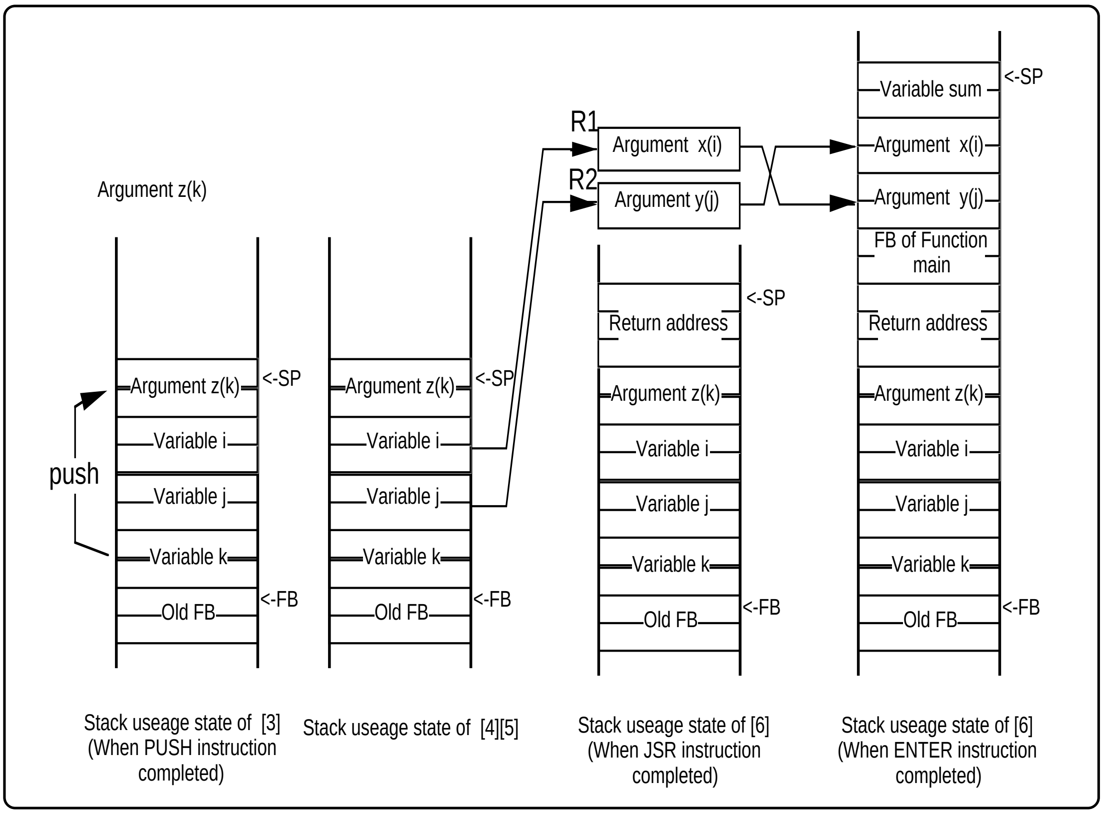


Figure D.18 Calling Function func and Entry Processing

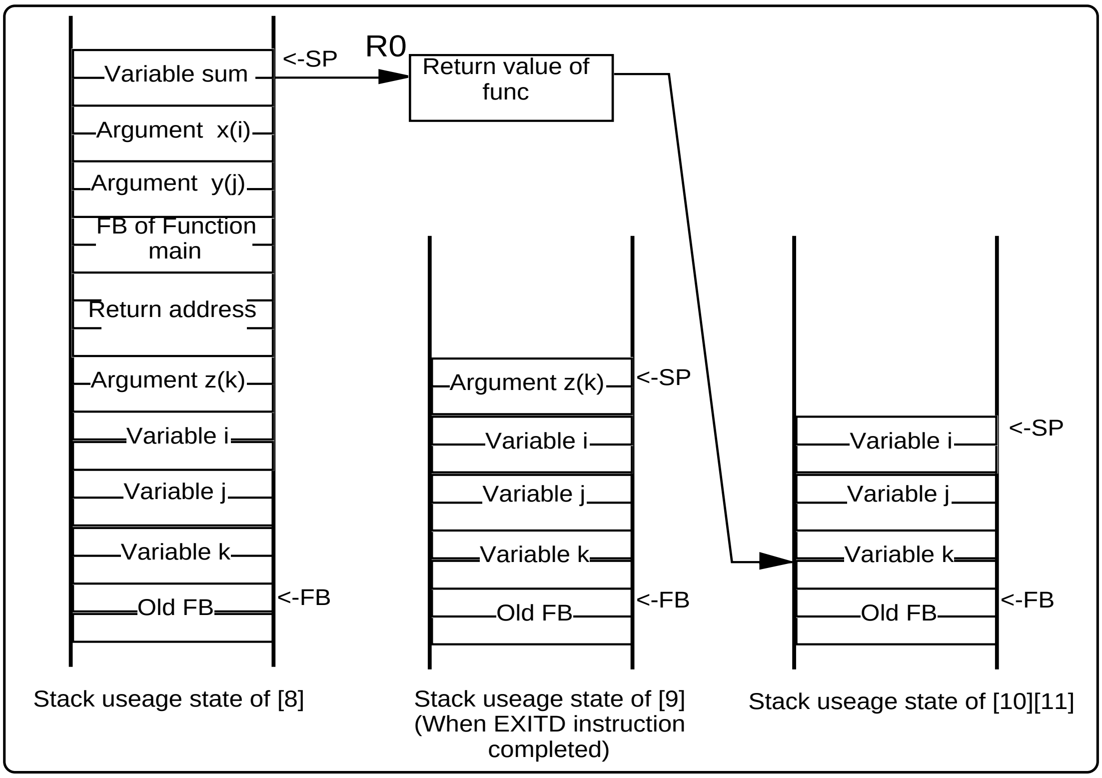


Figure D.19 Exit Processing of Function func

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.

```
func()
{
    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

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.
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	Defines the FILE structure.
	Defines a stream name.
	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 long-type 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 int-type 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.

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	Contents
Initialize	init	Initializes M16C/60 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	Function	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 upper-case 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

E.2.1 Overview of Standard Library

NC30 has 119 Standard Library items. Each function can be classified into one of the following 11 categories according to its function.

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

Functions for calculating sines (sin) and cosines (cos), etc.

* 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 ❍ in the table). However, if the function does not have reentrancy (indicated by a ✕ 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 lower-case.	❍
toupper	Converts the character from a lower-case to an upper-case.	❍

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.	×
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.	×
Determi-	ferror	Checks input/output errors.	×
nation	feof	Checks EOF (End of File).	×

d. Memory Management Functions

The following lists memory management functions.

Table E.5   Memory Management Functions

Function	Contents	Reentrant
calloc	Allocates a memory area and initializes it to zero (0).	×
free	Frees the allocated memory area.	×
malloc	Allocates a memory area.	×
realloc	Changes the size of an allocated memory area.	×

e. Memory Handling Functions

The following lists memory handling functions.

Table E.6 Memory Handling Functions

Type	Function	Contents	Reentrant
Initialize	bzero	Initializes a memory area (by clearing it to zero).	❍
Copy	bcopy	Copies characters from a memory area to another.	❍
	memcpy	Copies characters ('n' bytes) from a memory area to another.	❍
	memset	Set a memory area by filling with characters.	❍
Compare	memcmp	Compares memory areas ('n' bytes).	❍
	memicmp	Compares memory areas (with alphabets handled as upper-case letters).	❍
Move	memmove	Moves the area of a character string.	❍
Search	memchr	Searches a character from a memory area.	❍

f. Execution Control Functions

The following lists execution control functions.

Table E.7 Execution Control Functions

Function	Contents	Reentrant
abort	Terminates the execution of the program.	❍
longjmp	Performs a global jump.	❍
setjmp	Sets a stack environment for a global jump.	❍

g. Mathematical Functions

The following lists mathematical functions.

Table E.8 Mathematical Functions

Function	Contents	Reentrant
acos	Calculates arc cosine.	❍
asin	Calculates arc sine.	❍
atan	Calculates arc tangent.	❍
atan2	Calculates arc tangent.	❍
ceil	Calculates an integer carry value.	❍
cos	Calculates cosine.	❍
cosh	Calculates hyperbolic cosine.	❍
exp	Calculates exponential function.	❍
fabs	Calculates the absolute value of a double-precision floating-point number.	❍
floor	Calculates an integer borrow value.	❍
fmod	Calculates the remainder.	❍
frexp	Divides floating-point number into mantissa and exponent parts.	❍
labs	Calculates the absolute value of a long-type integer.	❍
ldexp	Calculates the power of a floating-point number.	❍
log	Calculates natural logarithm.	❍
log10	Calculates common logarithm.	❍
modf	Calculates the division of a real number into the mantissa and exponent parts.	❍
pow	Calculates the power of a number.	❍
sin	Calculates sine.	❍
sinh	Calculates hyperbolic sine.	❍
sqrt	Calculates the square root of a numeric value.	❍
tan	Calculates tangent.	❍
tanh	Calculates hyperbolic tangent.	❍

h. Integer Arithmetic Functions

The following lists integer arithmetic functions.

Table E.9 Integer Arithmetic Functions

Function	Contents	Reentrant
abs	Calculates the absolute value of an integer.	❍
bsearch	Performs binary search in an array.	❍
div	Divides an int-type integer and calculates the remainder.	❍
labs	Calculates the absolute value of a long-type integer.	❍
ldiv	Divides a long-type integer and calculates the remainder.	❍
qsort	Sorts elements in an array.	❍
rand	Generates a pseudo-random number.	❍
srand	Imparts seed to a pseudo-random number generating routine.	❍

i. Character String Value Convert Functions

The following lists character string value convert functions.

Table E.10 Character String Value Convert Functions

Function	Contents	Reentrant
atof	Converts a character string into a double-type floating-point number.	❍
atoi	Converts a character string into an int-type integer.	❍
atol	Converts a character string into a long-type integer.	❍
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.	❍

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.

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 the range 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 an long-type integer.

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

●No value is returned.

[ReturnValue] ●Copies the number of bytes specified in size from the beginning of the area specified in src to the area specified in dtop.

[Description]

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 *s; ..... Search key
const void _far *s; ..... 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

[Argument]
char _far *top; .... Start address of the memory area to be cleared to zero
unsigned long size; ... 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.

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

[Argument] 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.

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.

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.

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:
  .... new line character ('\n')
  .... n-1 characters
  .... 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.
- Interprets code 0x1A as the end code and ignores any subsequent data.
- 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.

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.

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.

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);
struct lconv * localeconv(void); [NC308 only]

[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 x.

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 "value" is returned as the result of setjmp. However, if "val" is "0", it is converted to "1".

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

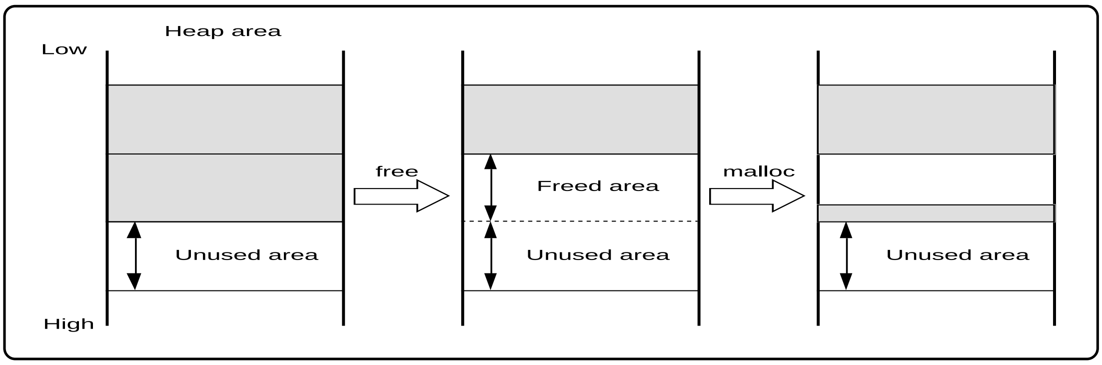
[ReturnValue] ● Returns NULL if a memory area of the specified size could not be allocated.

[Description] ● Dynamically allocates memory areas

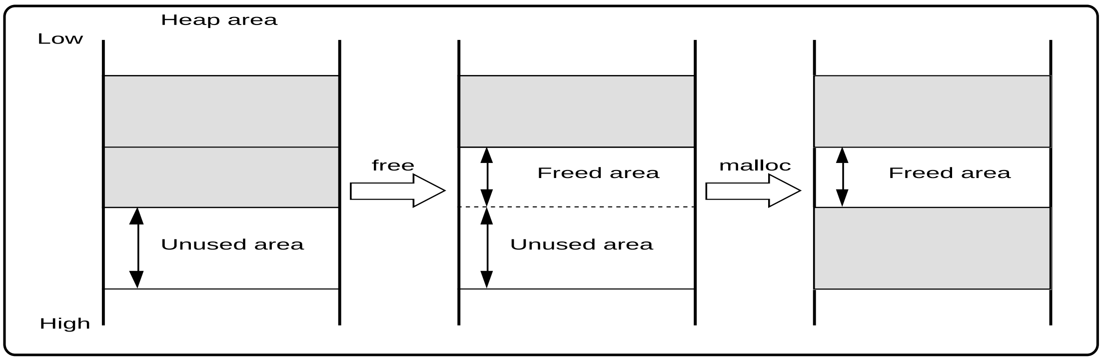
[Rule] ● malloc performs the following two checks to secure memory in the appropriate location.

(1)If memory areas have been freed with free

(1-1)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.



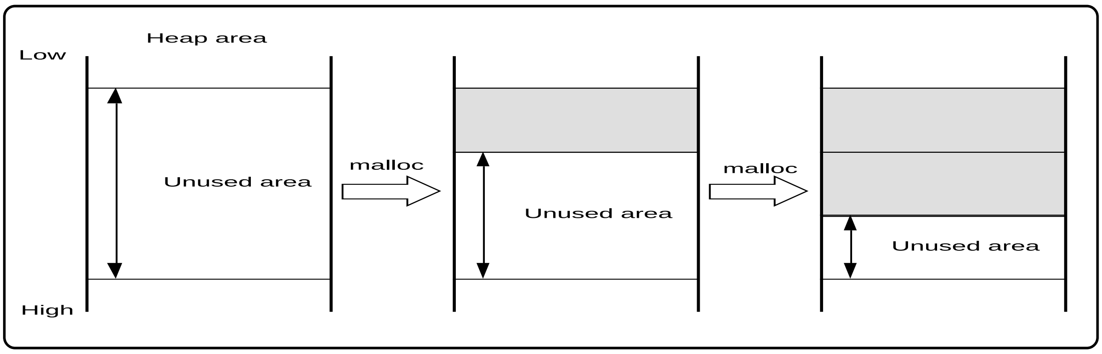
(1-2)If the amount of memory to be secured is larger than that freed, the area is secured from the lowest address of the unused memory toward the high address.



malloc

(2)If no memory area has been freed with free

(2-1)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.



(2-2)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.
- 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, -OR, or -OS, 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 *s1;	Pointer to the first memory area to be compared [NC308 only]
const void _far *s2;	Pointer to the second memory area to be compared
const void *s2;	Pointer to the second memory area to be compared [NC308 only]
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, -OR, or -OS, 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);
void * memcpy(s1, s2, n); [NC308 only]

[Method] function

[Argument]

void _far *s1; ...	Pointer to the memory area to be copied to
const void _far *2;	Pointer to the memory area to be copied from
size_t n;	Number of bytes to be copied

[ReturnValue]
- Returns the pointer to the memory area to which the characters have been copied.

[Description]
- Copies "n" bytes from memory "S2" to memory "S1".
- When you specify options -O, -OR, or -OS, 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, -OR, or -OS, 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 _far *s1;	Pointer to be moved to
const void _far *s2;	Pointer to be moved from
size_t n;	Number of bytes to be moved

[ReturnValue]
- Returns a pointer to the destination of movement.
- When you specify options -O, -OR, or -OS, 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>

char _far* memset(s, c, n);

[Method] function

[Argument]
void _far *s; ...... Pointer to the memory area to be set at
int c; .... Data to be set
size_t n; .... Number of bytes to be set

[ReturnValue]
- Returns the pointer to the memory area which has been set.

[Description]
- Sets "n" bytes of data "c" in memory "s".
- When you specify options -O, -OR, or -OS, 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 _far *pd; .. Pointer to an area for storing an integer

[ReturnValue] ● Returns the decimal part of a real number.

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; .... multiplier

[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.

Specifying format in printf-format

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.

● 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.

---

*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.

Specifying format in printf-form

2.Flags

● –

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.

3.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.

4.Precision

Specify a positive integer after '.'. If you specify only '.' with no value, it is interpreted as zero. The function and default value differs according to the conversion type.

Floating point type data is output with a precision of 6 by default. However, no decimal places are output if you specify a precision of 0.

●d, i, o, u, x, and X conversion

a. If the number of columns in the result of conversion is less than the specified number, the beginning is padded with zeros.

b. If the specified number of columns exceeds the minimum field width, the specified number of columns takes precedence.

c. 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 been processed.

d. The default is 1.

e. Nothing is output if zero with converted by zero minimum columns.

Specifying format in printf-form

- s conversion
  a. Represents the maximum number of characters.
  b. If the result of conversion exceeds the specified number of characters, the remainder is discarded.
  c. There is no limit to the number of characters in the default.
  d. If you specify an asterisk (*) for the precision, the integer of the parameter specifies the precision.
  e. 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.

5.I, 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.
- 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]

*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').

qsort

Integer Arithmetic Functions

[Function] Sorts elements in an array.

[Format] #include <stdlib.h>

void _far 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.

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".

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] 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 (l, 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.

Specifying format in scanf-form

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, and G

Convert to floating point format. If you specify modifier l, the target parameter must be a pointer to a double type. The default is a pointer to a float type.

Specifying format in scanf-form

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.l, L or h

- l: 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.
- If l 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] 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, -OR, or -OS, the system may selects another functions with good code efficiency by optimization.

strchr

String Handling Functions

[Function] Searches the specified character beginning with the top of the character string.

[Format] #include <string.h>

char _far * strchr(s, c);

[Method] function

[Argument] const char _far *s; Pointer to the character string to be searched in
int c; .... Character to be searched for

[ReturnValue]
- ● Returns the position of character "c" that is first encountered in character string "s."
- ● Returns NULL when character string "s" does not contain character "c".

[Description]
- ● Searches for character "c" starting from the beginning of area "s".
- ● You can also search for '\0'.
- ● When you specify options -O, -OR, or -OS, the system may selects another functions with good code efficiency by optimization.

*1. There must be adequate space to accommodate s1 plus s2.

strcmp

String Handling Functions

[Function] Compares character strings .

[Format] #include <string.h>

int strcmp(s1, s2);

[Method] 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]
- Compares each byte of two character strings ending with NULL
- When you specify options -O, -OR, or -OS, 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, -OR, or -OS, 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] function

[Argument]

char _far *s1; ...	Pointer to the character string to be copied to
const char _far *s2;	Pointer to the character string to be copied from

[ReturnValue] ● Returns a pointer to the character string at the destination of copy.

[Description]
- ● Copies character string "s2" (ending with NULL) to area "s1"
- ● After copying, the character string ends with NULL.
- ● When you specify options -O, -OR, or -OS, 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.

[Note]
- 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, -OR, or -OS, 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, -OR, or -OS, 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, -OR, or -OS, 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, -OR, or -OS, 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, -OR, or -OS, 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, -OR, or -OS, 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 other 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, -OR, or -OS, 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, -OR, or -OS, 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 **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, -OR, or -OS, 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]
- Returns the location (pointer) of the first character string "s2" from the beginning of area "s1".
- 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, -OR, or -OS, 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, -OR, or -OS, 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, -OR, or -OS, 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, -OR, or -OS, the system may selects another functions with good code efficiency by optimization.

tan

Mathematical Functions

[Function] Calculates tangent.

[Format] #include <math.h>

double _far 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.

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.

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.

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 );
size_t  wcstombs( s,wcs,n );          [NC308 only]
```

[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[3-5], -OS, or -OR, 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 in Table 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);

(2)Selection of high-speed library (NC30 only)

Some standard library functions have a pointer in argument. NC30 normally handles such pointers as the far pointer. For this reason, NC30 does not generate efficient code if the argument is a near pointer. Therefore, if the argument is a near pointer, the system performs optimization to choose a library function provided for use as near. The table below lists the functions that are subject to such optimization.

Table E.14 Library Functions Subject to Optimization

Function Name	Function Name	Function Name	Function Name
bcopy	strcat	strnicmp	strstr
bzero	strchr	strlen	strspn
memchr	strcmp	strncat	strtod
memcmp	strcoll	strncmp	strtok
memcpy	strcpy	strncpy	strtol
memicmp	strcspn	strnicmp	strtoul
memmove	strerror	strpbrk	strxfrm
memset	stricmp	strrchr	

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 low-level I/O functions.

In M16C/60 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.

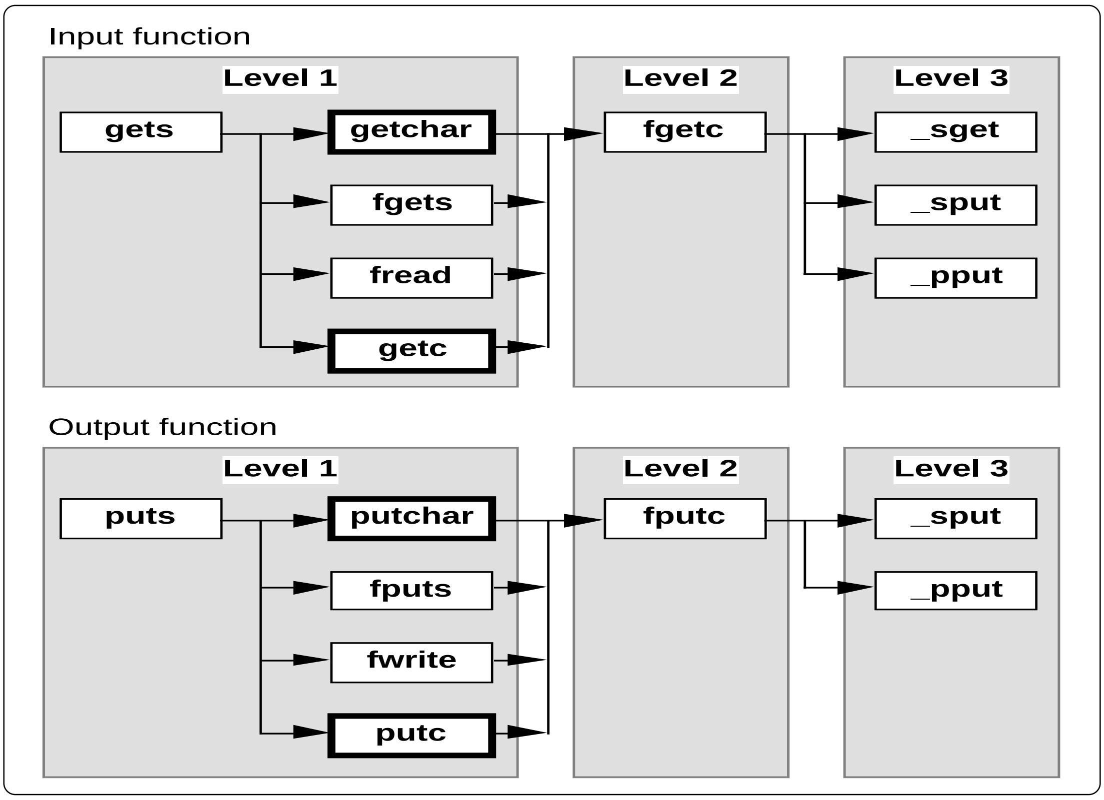


Figure E.1 Calling Relationship of I/O Functions

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.

a. Modify the level 3 I/O function(s)

b. Set the stream

c. Compile the modified source program(s)

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/60 series 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.

[Circuit settings]

- ●Processor mode: Microprocessor mode
- ●Clock frequency: 20MHz
- ●External bus size: 16 bits

[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.13 lists the specifications of these functions.

Table E.13 Specifications of Level 3 Functions

Input functions	Parameters	Return value (int type)
_sget	None.	If no error occurs, returns the input character Returns EOF if an error occurs
_sput		
_pput		

Output functions	Parameters (int type)	Return value (int type)
_sput	Character to output	If no error occurs, returns 1 Returns EOF if an error occurs
_pput		

Serial communication is set to UART1 in the M16C/60 series's 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 1

Specify these commands at the beginning of device.c, or specify following option, when compiling.

●To use UART0 .... -DUART0

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 EQU 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.

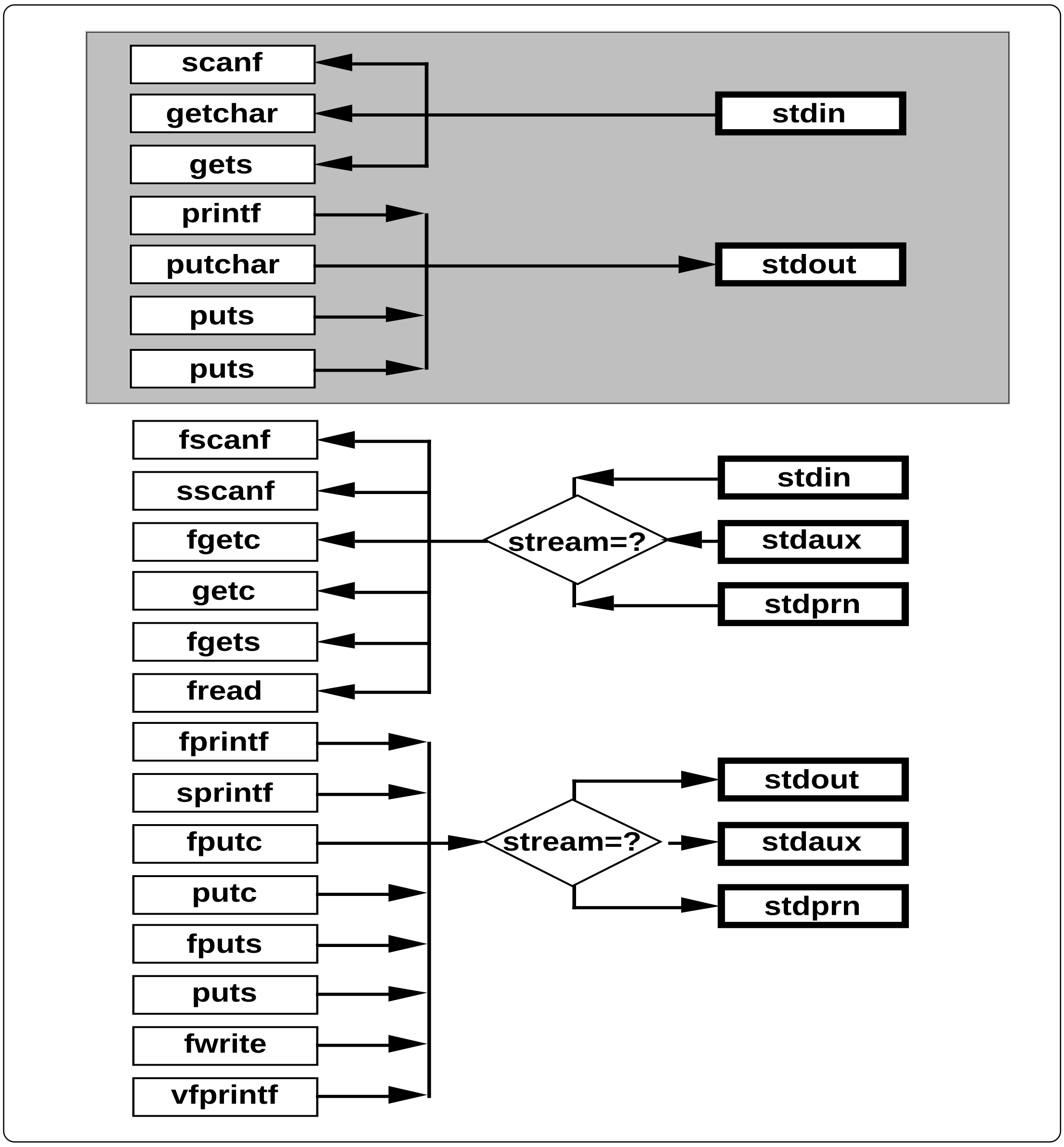


Figure E.3   Relationship of Functions and Streams

Figure E.4 shows the stream definition in stdio.h.

```
/*******************************************************************************
*
* standard I/O header file
          :
      (omitted)
          :
typedef struct _iobuf {
        char _buff;             /* Store buffer for ungetc */                ⇐[1]
        int _cnt;               /* Strings number in _buff(1 or 0) */        ⇐[2]
        int _flag;              /* Flag */                                   ⇐[3]
        int _mod;               /* Mode */                                   ⇐[4]
        int (* _func_in)(void);     /* Pointer to one byte input function */  ⇐[5]
        int (* _func_out)(int);     /* Pointer to one byte output function */ ⇐[6]
} FILE;
#define _IOBUF_DEF
          :
      (omitted)
          :
extern FILE _iob[];
#define  stdin   (&_iob[0])         /* Fundamental input */
#define  stdout  (&_iob[1])         /* Fundamental output */
#define  stdaux  (&_iob[2])         /* Fundamental auxialiary input output */
#define  stdprn  (&_iob[3])         /* Fundamental printer output */

#define  stderr  stdout

/**********************************************************
**********************************************************/
#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
    speed(_96, _B8, _PN, _S2);
#endif
    init_prn();
}
```

Figure E.5   Source file of init function (init.c)

In systems using the two M16C/60 series 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)
          :
    speed0(_96, _B8, _PN, _S2);      ⇐[4](set UART0 speed function)
    speed(_96, _B8, _PN, _S2);
    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 NC30, and their countermeasures.

F.1 Message Format

If, during processing, NC30 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 nc30 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

The following pages list the error messages and their countermeasures. cpp30 messages are listed according to their Nos. The messages output by other programs are listed alphabetically (symbols followed by letters).

*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

Tables F.1 and F.2 list the nc30 compile driver error messages and their countermeasures.

Table F.1 nc30 Error Messages (1/2)

Error message	Description and countermeasure
Arg list too long	● The command line for starting the respective processing system is longer than the character string defined by the system. ⇒Specify a NC30 option to ensure that the number of characters defined by the system is not exceeded. Use the -v option to check the command line used for each processing block.
Cannot analyze error	● This error message is not normally displayed. (It is an internal 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 command file to 2048 max.
Core dump (*command-name*)	● The processing system (indicated in parentheses) caused a core dump. ⇒The processing system is not running correctly. Check the environment variables and the directory containing the processing system. If the processing system still does not run correctly, Please contact Renesas Solutions Corp.
Exec format error	● Corrupted processing system executable file. ⇒Reinstall the processing system.
Ignore option '-?'	● You specified an illegal option (-?) for NC30. ⇒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	● This error message is not normally displayed. (It is an internal error.) ⇒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.

Table F.2 nc30 Error Messages (2/2)

Error message	Description and countermeasure
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.
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	[UNIX]: ● Insufficient swap area ⇒Increase the swap area by, for example, adding a secondary swap area. [MS-Windows 95,98 / NT]: ● 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.) ⇒ Contact Renesas Solutions Corp.
Result too large	● This error message is not normally displayed. (It is an internal error.) ⇒Contact Renesas Solutions Corp.
Too many open files	● This error message is not normally displayed. (It is an internal error.) ⇒Contact Renesas Solutions Corp.

F.3 cpp30 Error Messages

Tables F.3 to F.6 list the error messages output by the cpp30 preprocessor and their countermeasures.

Table F.3 cpp30 Error Messages (1/4)

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.

Table F.4　cpp30 Error Messages (2/4)

No.	Error message	Description and countermeasure
16	cannot write output file	● Error writing to output file. ⇒Check the available space on disk.
17	input file name buffer over-flow	● 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 identifier	● Insufficient memory for macro name and contents of macro [UNIX]: ⇒Increase the swap area [MS-Windows]: ⇒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 (8). ⇒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 - #as-sert operation expression. ⇒Rewrite operation expression correctly.
27	macro argument error	● Error in number of macro parameters when ex-panding 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 con-verting macros. ⇒Reduce each line in the source file to a maximum of 1023 characters. If you anticipate macro con-version, 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.

Table F.5 cpp30 Error Messages (3/4)

No.	Error message	Description and countermeasure
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.
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 error	● #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.

Table F.6 cpp30 Error Messages (4/4)

No.	Error message	Description and countermeasure
61	illegal line command usage	● Error in #line. ⇒Correct the source file.

F.4 cpp30 Warning Messages

Table F.7 shows the warning messages output by cpp30 and their countermeasures.

Table F.7 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.

F.5 ccom30 Error Messages

Tables F.8 to F.20 list the ccom30 compiler error messages and their countermeasures.

Table F.8 ccom30 Error Messages (1/14)

Error message	Description and countermeasure
#pragma *PRAGMA-name function-name* redefined	● The same function is defined twice in #pragma-name. ⇒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 PARAMETER & function prototype mismatched	● The function specified by #pragma PARAMETER does not match the contents of argument in prototype declaration. ⇒Make sure it is matched to the argument in prototype declaration.
#pragma PARAMETER's function argument is struct or union	● The struct or union type is specified in the prototype declaration for the function specified by #pragma PARAMETER. ⇒Specify the int or short type, 2-byte pointer type, or enumeration type in the prototype declaration.
#pragma PARAMETER must be declared before use	● A function specified in the #pragma PARAMETER declaration is defined after call for that function. ⇒Declare a function before calling it.
#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 argument	● An appropriate storage class is used in argument declaration. ⇒Use the correct storage class.

Table F.9　ccom-mocc Error Messages (2/14)

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 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 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 permission of the file.
can't open *file-name*	● The file cannot be opened. ⇒Check the permission of the file.

Table F.10 ccom30 Error Messages (3/14)

Error message	Description and countermeasure
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.
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 do-while statement.
do while (void) statement	● The void type is used in the expression of the do-while statement. ⇒Use the scalar type for an expression in the do-while statement.
duplicate frame position defind *variable-name*	● Auto variable is described more than twice. ⇒Write the type specifier correctly.
duplicate 'long'	● long 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.

Table F.11 ccom30 Error Messages (4/14)

Error message	Description and countermeasure
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.
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.

Table F.12 ccom30 Error Messages (5/14)

Error message	Description and countermeasure
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(*tag-name*)'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'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 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.

Table F.13 ccom30 Error Messages (6/14)

Error message	Description and countermeasure
invalid '()' operands	● 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 expression is not pointer type. ⇒For a multiplication, check the expressions on the left and right sides of the operator. For a pointer, check the type of the right-side expression.
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 '=' 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.

Table F.14 ccom30 Error Messages (7/14)

<table>
<tr><th>Error message</th><th>Description and countermeasure</th></tr>
<tr><td>invalid '>>=' operands</td><td>● The >>= operation contains an error.
⇒Check the expressions on the left and right sides of the operator.</td></tr>
<tr><td>invalid '[ ]' operands</td><td>● 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 [ ].</td></tr>
<tr><td>invalid '^=' operands</td><td>● The ^= operation contains an error.
⇒Check the expressions on the left and right sides of the operator.</td></tr>
<tr><td>invalid ' | =' operands</td><td>● The | = operation contains an error.
⇒Check the expressions on the left and right sides of the operator.</td></tr>
<tr><td>invalid ' || ' operands</td><td>● The || operation contains an error.
⇒Check the expressions on the left and right sides of the operator.</td></tr>
<tr><td>invalid '%=' operands</td><td>● The %= operation contains an error.
⇒Check the expressions on the left and right sides of the operator.</td></tr>
<tr><td>invalid ++ operands</td><td>● The ++ unary operator or postfix operator contains an error.
⇒ For the unary operator, check the right-side expression. For the postfix operator, check the left-side expression.</td></tr>
<tr><td>invalid - - operands</td><td>● The -- unary operation or postfix operation contains an error.
⇒For the unary operator, check the right-side expression. For the postfix operator, check the left-side expression.</td></tr>
<tr><td>invalid -> used</td><td>● The left-side expression of -> is not struct or union.
⇒The left-side expression of -> must have struct or union.</td></tr>
<tr><td>invalid (? :)'s condition</td><td>● The ternary operator is erroneously written.
⇒Check the ternary operator.</td></tr>
<tr><td>Invalid #pragma OS Extended function interrupt number</td><td>● The INT No. in #pragma OS Extended function is invalid.
⇒Specify correctly.</td></tr>
<tr><td>Invalid #pragma INTCALL interrupt number</td><td>● The INT No. in #pragma INTCALL is invalid.
⇒Specify correctly.</td></tr>
<tr><td>Invalid #pragma SPECIAL page number</td><td>● The No. in #pragma SPECIAL is invalid.
⇒Specify correctly.</td></tr>
<tr><td>invalid CAST operand</td><td>● 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.</td></tr>
</table>

Table F.15 ccom30 Error Messages (8/14)

Error message	Description and countermeasure
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, do-while, and for.
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.

Table F.16 ccom30 Error Messages (9/14)

Error message	Description and countermeasure
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.
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.

Table F.17 ccom30 Error Messages (10/14)

Error message	Description and countermeasure
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 types 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.
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	[UNIX version] ● The swap area is insufficient. ⇒Increase the swap area. [MS-Windows 95,98 / NT version] ● The memory area is insufficient. ⇒Increase the memory or the swap area.

Table F.18 ccom30 Error Messages (11/14)

Error message	Description and countermeasure
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 *variable-name*	● 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.
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.

Table F.19 ccom30 Error Messages (12/14)

Error message	Description and countermeasure
Sorry, stack frame memory exhaust, max. 128 bytes but now *nnn* bytes(NC30, NC308 only)	● 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, stack frame memory exhaust, max. 64(or 255) bytes but now *nnn* bytes	● The stack frame maximum is follows. 64 bytes (NC79) 255bytes (NC30, NC77 and NC79 with -fDPO8 option used) Currently nnn bytes have been used. ⇒Reduce the auto variables, parameters, and other variables stored in the stack frame area.
Sorry, compilation terminated because of these errors in *function-name*.	● 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.
System Error	● It does not normally occur. (This is an internal error.)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.

Table F.20 ccom30 Error Messages (14/14)

Error message	Description and countermeasure
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.
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.

Table F.21 ccom30 Error Messages (13/13)

Error message	Description and countermeasure
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	● A pointer variable decleared by #pragma EXT4MPTR is duplecate. ⇒ Declare the variable only one time.
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 ccom30 Warning Messages

Tables F.21 to F.30 list the ccom30 compiler warning messages and their countermeasures.

Table F.21 ccom30 Warning Messages (1/10)

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 *pragma-name*, 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 *pragma-name*, 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 stÒ•ct, union, or double.
#pragma *pragma-name* unknown switch,ignored	● The switch specified in the #pragma *pragma-name* is invalid. ⇒ Write it correctly.

Table F.22 ccom30 Warning Messages (2/10)

Warning message	Description and countermeasure
#pragma ADDRESS variable initialized, ADDRESS ignored	● The variable specified in #pragma ADDRESS is initialized. The specification of #pragma ADDRESS will be nullified. ⇒ Delete either #pragma ADDRESS 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.
#pragma directive conflict	● #pragma of different functions is specified for one function. ⇒ Write it correctly.
#pragma DP[n]DATA format error,ignored (NC79 only)	● You have also specified option -fDPO8. ⇒ If you specify both #pragma DP[n]DATA and -fDPO8, #pragma DP[n]DATA is invalid. Delete the option -fDPO8.
	● You have made an error in the format of #pragma DP[n]DATA. ⇒ Correct the format.
#pragma JSRA illegal location, ignored (NC30,NC308 only)	● Do not put #pragma JSRA inside function scope. ⇒ Write #pragma JSRA outside a function.
#pragma JSRW illegal location, ignored (NC30,NC308 only)	● Do not put #pragma JSRW inside function scope. ⇒ Write #pragma JSRW outside a function.
#pragma PARAMETER function's address used	● The address of function specified #pragma PARAMETER is assigned to the pointer variable. ⇒ As don't assign, write correctly.
#pragma control for function duplicate, ignored (NC30,NC308 only)	● 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, TASK, HANDLER, CYCHANDLER, or ALMHANDLER.
'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.
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.

Table F.23 ccom30 Warning Messages (3/10)

Warning message	Description and countermeasure
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 511 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 always Bank 0 (NC77,NC79 only)	● Function specified in #pragma CYCHANDLER or ALMHANDLER are always compiled in bank 0 (addresses below 10000H). ⇒None.
cyclic or alarm handler always load DT (NC77,NC79 only)	● There is no need to #pragma LOADDT a function specified in #pragma CYCHANDLER or ALMHANDLER. ⇒Delete #pragma LOADDT.

Table F.24 ccom30 Warning Messages (4/10)

Warning message	Description and countermeasure
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.
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.
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 duplicate	● 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.

Table F.25 ccom30 Warning Messages (5/10)

Warning message	Description and countermeasure
illegal storage class for argument, 'extern' ignored	● An invalid storage class is used in the argument list of function definition. ⇒Specify the correct storage class.
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.
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.

Table F.26 ccom30 Warning Messages (6/10)

Warning message	Description and countermeasure
invalid asm's M flag (NC77,NC79 only)	● Error in M flag value in asm statement. ⇒Specify an integer constant (0, 1, or 2).
invalid asm's MX flag, ignored (NC77,NC79 only)	● Error in MX flag value in asm statement. ⇒ Specify an integer constant (0, 1, or 2).
invalid asm's X flag (NC77,NC79 only)	● Error in X flag value in asm statement. ⇒Specify an integer constant (0, 1, or 2).
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.
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.
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 non-prototyped 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 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.

Table F.27 ccom30 Warning Messages (7/10)

Warning message	Description and countermeasure
No initialized of *valiable 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 ë*variable name*í 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 -WNP 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.
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.

Table F.28 ccom30 Warning Messages (8/10)

Warning message	Description and countermeasure
RESTRECT 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.
sorry, get stack's address, but DT not 0 (NC77,NC79 only)	● This error occurs when the -bank option is specified. When the address of an auto variable is assigned to a pointer and an object referenced using that pointer, DT points to outside bank 0, preventing bank 0 from being referenced. ⇒Declare the variable as a far type.
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 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 void	● An attempt is made to find the size of void. This is an invalid size. ⇒The size of void cannot be found.
standard libraly ì*function-name*()î need ì*include-file name*î	● This standard library function is used without its header file included. ⇒Be sure to include the header file.
static valuable 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.
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.

Table F.29 ccom30 Warning Messages (9/10)

Warning message	Description and countermeasure
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 it 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 that has a default argument. ⇒ Declare the default argument before using a function.
this interrupt function is called as normal function before	● A function once used is declared in #pragma INTERRUPT. ⇒ An interrupt function cannot be called. Check the content of #pragma.
too big octal character	● The character constant or the octal constant in the character string exceeded the limit value (255 in decimal). ⇒ Do not use a value greater than 255 to describe the constant.
too few parameters	● Arguments are insufficient compared to the number of arguments declared in prototype declaration. ⇒ Check the number of arguments.
too many parameters	● Arguments are excessive compared to the number of arguments declared in prototype declaration. ⇒ Check the number of arguments.
uncomplete struct member	● An incomplete structure is written as a member. ⇒ Write a complete structure.
unknown #pragma STRUCT xxx	● #pragma STRUCTxxx cannot be processed. This 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.
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.

Table F.30 ccom30 Warning Messages (10/10)

Warning message	Description and countermeasure
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 initialized by a character string of the char type. ⇒ Make sure that the types of the initialize expression are matched.
zero divide in constant folding	● The divisor in the divide operator or remainder calculation operator is 0. ⇒ Use any value other than 0 for the divisor.
zero divide, ignored	● The divisor in the divide operator or remainder calculation 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.

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.
(This utility is not included in the entry version.)

G.1 Introduction of utl30

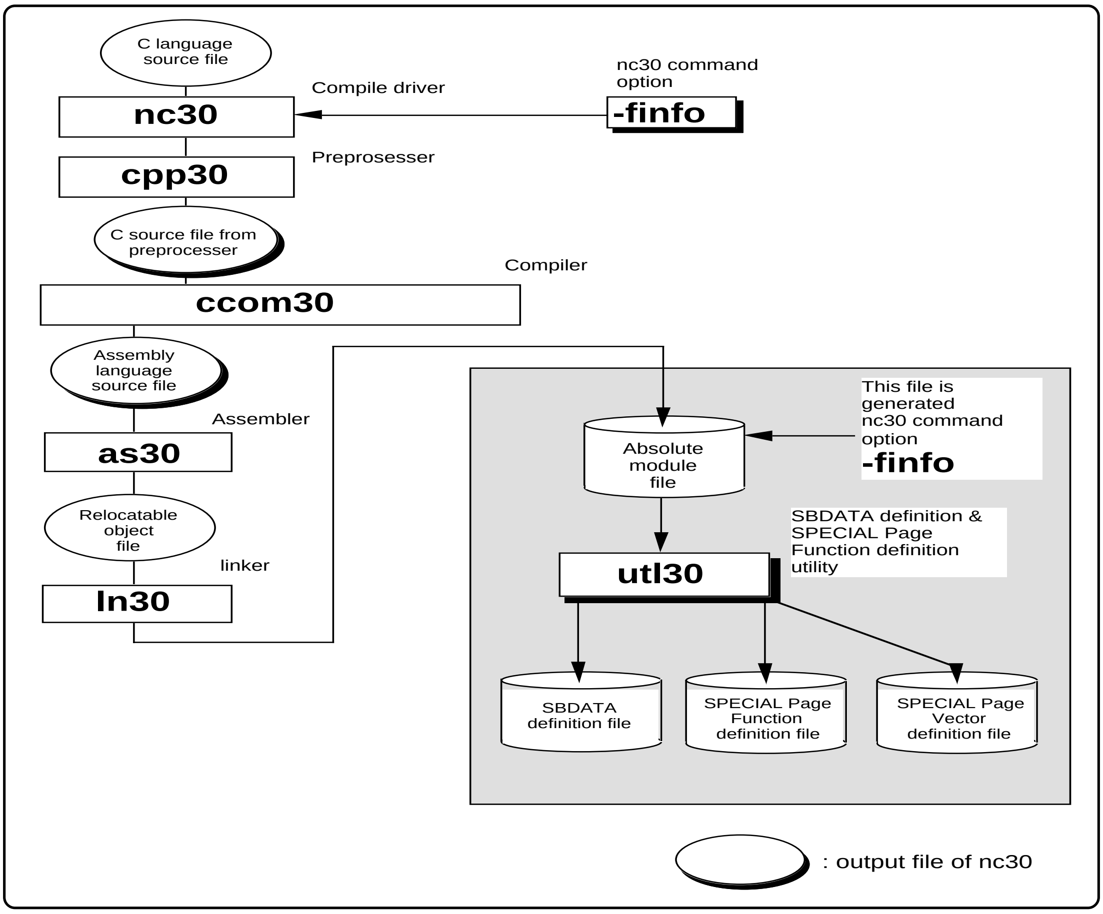
G.1.1 Introduction of utl30 processes

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,î#pragma SBDATAî) and a file that contains SPECIAL page function declarations (located in the SPECIAL page area beginning with the most frequently used one,î#pragma SPECIALî).

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.



FigureG.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.

% utl30Δ[*command-line-option*]Δ<*map-file-name*>

% :Prompt
< > :Mandatory item
[ ] :Optional item
Δ :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)

● Output the absolute module file

```
%nc30 ncrt0.a30 -finfo sample.c<RET>
M16C/60 NC30 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.x30
sample.c
```

● Output SBDATA declaration

```
%utl30 -sb30 ncrt0.x30 -o sample<RET>
M16C/60 UTILITY UTL30 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED


%
```

● Output SPECIAL page Function declaration

```
%utl30 -sp30 ncrt0.x30 -o sample<RET>
M16C/60 UTILITY UTL30 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY 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"

Table 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
-sb30 - 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.
-o<*function name*>	None.	Outpus 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.
-fover_write	-fOW	Forcibly writes over the output file name specified with the -o option.
-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.
-Wstdout	None.	Output the warning and error messages to the honst machines standard output device.
-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.
-fsection	None.	The variables and functions specified by #pragma SECTION are also included among those to be processed.

-sb30

Outputs SBDATA declaration

Function : Outputs SBDATA declaration. This option can be specified simultaneously with -sp30.

Execution example :

```
% utl30 -sb30 ncrt0.x30 -o sample
M16C/60 UTILITY UTL30 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%
```

-sp30

Outputs SPECIAL page function declaration

Function : Outputs SPECIAL page function declaration. This option can be specified simultaneously with -sb30.

Execution example :

```
% utl30 -sp30 ncrt0.x30 -o sample
M16C/60 UTILITY UTL30 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%
```

-o

Outputs the declared SBDATA result display file

Function : Outpus 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.

Execution example :

● Output SBDATA declaration

```
% utl30 -sb30 ncrt0.x30 -o sample
M16C/60 UTILITY UTL30 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%type sample.h
/*
* #pragma SBDATA Utility
*/
/* SBDATA Size [255] */
#pragma SBDATA  z     /* size = (2) / ref=[2] */
    (omit)
#pragma SBDATA  vx    /* size = (2) / ref=[1] */

%
```

● Output SPECIAL page Function declaration

```
% utl30 -sp30 ncrt0.x30 -o sample
M16C/60 UTILITY UTL30 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%type sample.h
/*
* #pragma SPECIAL PAGE Utility
*/

#pragma SPECIAL 255 func()       /* size = (200) / ref=[2] */
    (omit)
#pragma SPECIAL 254 func1()      /* size = (200) / ref=[1] */

%
```

-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.

Execution example :

● Output SBDATA declaration

```
% utl30 -sb30 -all ncrt0.x30
M16C/60 UTILITY UTL30 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%type sample.h
/*
* #pragma SBDATA Utility
*/
/* SBDATA Size [255] */
#pragma SBDATA  z     /* size = (2) / ref=[2] */
    (omit)
#pragma SBDATA  vx    /* size = (2) / ref=[1] */

%
```

● Output SPECIAL page Function declaration

```
% utl30 -sp30 -all ncrt0.x30
M16C/60 UTILITY UTL30 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%type sample.h
/*
* #pragma SPECIAL PAGE Utility
*/

#pragma SPECIAL 255 func()       /* size = (200) / ref=[2] */
    (omit)
#pragma SPECIAL 254 func1()      /* size = (2000) / ref=[1] */

%
```

Supplement:: 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.

-Wstdout

warning option

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

Execution example :

```
% utl30 -o sample ncrt0.x30 -Wstdout
M16C/60 UTILITY UTL30 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

warning:cannot open file 'ncrt0.x30'

%
```

-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.

Execution example :

♦ To specify a single number.

-SP=<*number*>

Example) %utl30 -sp30 -sp=255 ncrt0.x30

♦ To specify two or more numbers.

-SP=<*number*>,<*number*>,...

Example) %utl30 -sp30 -sp=255,254 ncrt0.x30

♦ To specify a range of numbers.

-SP=<*number*> **-** <*number*>

Example) %utl30 -sp=255-250 ncrt0.x30

-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.

-fover_write -fOW

Outputs SBDATA declaration or SPECIAL function declaration to a file

Function : Does not check whether the output file specified by -? already exists. If such file exists, it is overwritten.
This option must be specified along with the -? option.

G.3 Notes

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.

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

Only global variables are valid in using utl30

Types of variables are as follows.

(1variables of _Bool

(2)variables of unsigned char and signed char type

(3)variables of unsigned short and signed short type

(4)variables of unsigned int and signed int type

(5)variables of unsigned long and signed long type

(6)variables of unsigned long long and signed long long type

Variables give below are excluded from SBDATA declaration.

(1)variables positioned in sections worked on by #pragma SECTION

(2)variables defined by #pragma ADDRESS

(3)variables defined by #pragma ROM

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.

(1)Functions which are not declared with static

(2)Functions which are called three times or more

Note, however, that even the above functions may not be processed if they belong to one of the following:

(1)functions positioned in sections worked on by #pragma SECTION

(2)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. Fig. G.4 shows an example of making entries in utl30 , and Fig.G.5 shows an example of SBDATA declaration file.

```
% utl30 ncrt0.x30 -osbdata<RET>
```

%	: Prompt
ncrt0.x30	: Name of map 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] */
/*
* End of File
*/
                                  Size=()   is  size of data
                                  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 .Fig.G.6 shows an example of making setting in a SBDATA file.

```
#include  "sbdata.h"

func()
{
    (ommit)
```

Figure G.6 Example of making settings in a SBDATA

b. Adjustment in an instance in which SB declaration is made in asesembler

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
    (ommit)
    .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)
                .
                .
//#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 map 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 SPECIAL 253 func3        /* size = (100) / ref = [    5] */
/*
* End of File
*/
```

Figure G.9 SPECIAL Page Function declaration File (special.h)

```
;
;    #pragma SPECIAL PAGE Utility
;
;    special page definition
;
SPECIAL .macro NUM
        .org     0FFFFEH-(NUM*2)
        .glb     __SPECIAL_@NUM
        .word    __SPECIAL_@NUM & 0FFFFH
.endm

        SPECIAL  255
        SPECIAL  254
        SPECIAL  253
;
;    End of File
;
```

Figure G.10 SPECIAL Page vector declaration File (special.inc)

You include the SPECIAL Page Finction declaration file generated above in a program as a header file . Fig.G.11 shows an example of making setting in a SPECIAL Page Function declaration File.

```
#include  "special.h"

func()
{
    (ommit)
```

Figure G.11 Example of making settings in a SPECIAL Page Function File

Includes, during startup, the SPECIAL Page vector definition file as a file to be included. Fig. G.12 shows an example of setting up a SPECIAL Page vector definition file.

```
    (ommit)

.section vector
.include "special.inc"

    (ommit)
```

Figure G.12 Example of making settings in a SPECIAL Page Function File for sect30.inc

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-name'	● 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-name'	● The function shown here is declared in multiple files with different storage classes, types, etc. ⇒Check how this function is declared.

MEMO

NC30 V.5.20 User’s Manual

Rev. 1.00
Sep 1, 2003
REJ10J0225-0100Z

COPYRIGHT ©2003 RENESAS TECHNOLOGY CORPORATION
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

Renesas Technology Corp.
2-6-2, Ote-machi, Chiyoda-ku, Tokyo, 100-0004, Japan