To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

RENESAS

User's Manual

R32C/100 Series
C Compiler Package V.1.01

C Compiler User's Manual

Renesas Electronics
www.renesas.com

Rev.1.00 2007.08

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 /tools

Contents

Preface

NC100 is the C compiler for the Renesas 32-bit microcomputer R32C/100 series. NC100 converts programs written in C into assembly language source files for the R32C/100 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 NC100.

- Microsoft, MS-DOS, Windows and Windows NT are either registered trademarks or trademarks or Microsoft Corporation in the United States and other countries. HP-UX is a registered trademark of Hewlett-Packard Company.
- 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
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- 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.
- HP9000 is a product name of Hewlett-Packard Company.
- SPARC and SPARCstation are registered trademarks of SPARC International, Inc.
- Intel and Pentium are registered trademarks of Intel Corporation.
- Adobe and Acrobat are registered trademarks of Adobe Systems Incorporated.
- Netscape and Netscape Navigator are registered trademarks of Netscape Communications Corporation in the U.S. and other countries.

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

Terminology

The following terms are used in the NC100 User Manuals.

Term	Meaning
NC100	Compiler package for R32C/100 series
nc100	Compile driver and its executable file
AS100	Assembler package included in Compiler package for R32C/100 series
as100	Relocatable macro assembler and its executable file
High-performance Embedded Workshop	Integrated development environment of attachment

Description of Symbols

The following symbols are used in the NC100 manuals.

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

Additional descriptions are provided where other symbols are used.

Chapter 1 Introduction to NC100

This chapter introduces the processing of compiling performed by NC100, and provides an example of program development using NC100.

1.1 NC100 Components

NC100 consists of the following five executable files:

(1) nc100 ....C Compile driver
(2) igen100 ....C Inline generator
(3) cpp100 ....C Preprocessor
(4) ccom100 ....C Compiler
(5) aopt100 ....Assembler optimizer
(6) Call Walker & gensni ....Stack analysis tool & Stack information analysis utility
(7) MapViewer ....Map Viewer

1.2 NC100 Processing Flow

Figure 1.1 illustrates the NC100 processing flow.

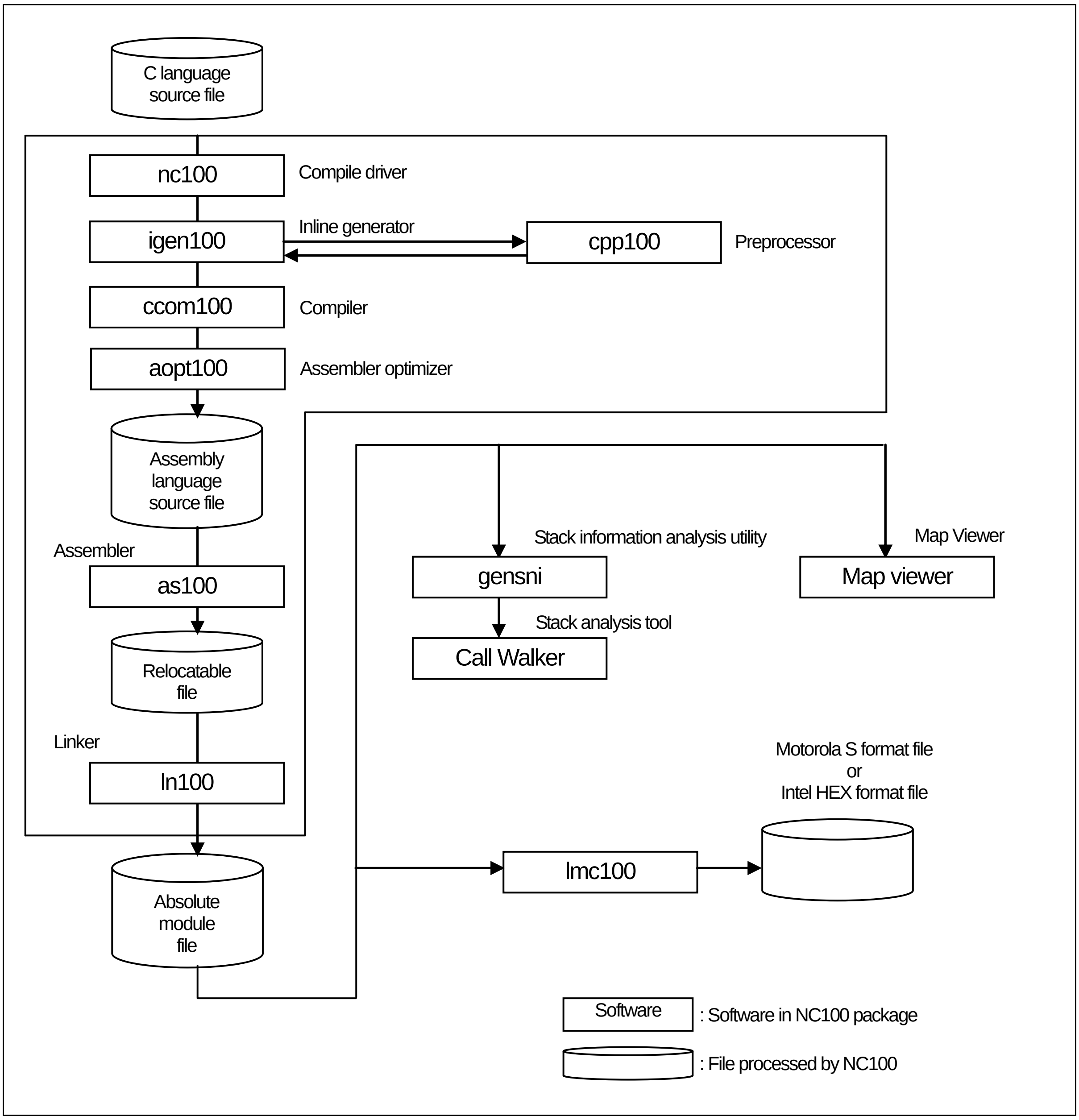


Figure 1.1 NC100 Processing Flow

1.2.1 nc100

nc100 is the executable file of the compile driver.
By specifying options, nc100 can perform the series of operations from compiling to linking. You can also specify for the as100 relocatable macro assembler and four for the ln100 linkage editor by including the –as100 and –ln100 command line options when you start nc100.

1.2.2 igen100

igen100 is the executable file of the inline generator.
igen100 calls cpp100.

1.2.3 cpp100

cpp100 is the executable file for the preprocessor.
cpp100 processes macros starting with # (#define, #include, etc.) and performs conditional compiling (#if-#else-#endif, etc.).

1.2.4 ccom100

ccom100 is the executable file of the compiler itself.
C source programs processed by cpp100 are converted to assembly language source programs that can be processed by as100.

1.2.5 aopt100

aopt100 is the assembler optimizer
It optimizes the assembler codes output by ccom100.

1.2.6 Call Walker & gensni

CallWalker is the utility to graphically display the relationship between stack sizes and function calls that is needed for program operation. Similarly, gensni is the utility to analyze the necessary information.
CallWalker loads a stack information file (.x30) that is output by gensni to display the amount of stacks used.
The amount of stacks used by an assembly program that cannot be output to a stack information file can be added or edited by using the editing facility, making it possible to find the total amount of stacks used in the entire system.
The edited information for the amount of stacks used can be saved or loaded as a call information file (*.cal).
Before CallWalker & gensni can be used, the compile driver's startup option -finfo must be specified during compilation so that inspector information will be added to the absolute module file (.x30).

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.

1.3 Notes

To use the technical contents shown in product data, diagrams or tables or the programs or algorithms presented herein for your system, please carefully evaluate their suitability as part of the entire system, not singly as technical content, program or algorithm alone, to determine in advance whether they are actually suitable for your system. Renesas Technology Corporation and Renesas Resolutions Corporation will not assume responsibility for the suitability of said items in user systems.

1.3.1 Notes about Version-up of compiler

The machine-language instructions (assembly language) generated by NC100 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 R32C'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.

```
#pragma  ADDRESS  TA0IC  006Ch          /* R32C/100 Timer A0 interrupt control register */

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

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

Figure 1.2 C language description to SFR area

1.4 Example Program Development

Figure 1.3 shows the flow for the example program development using NC100. The program is described below. (Items [1] to [4] correspond to the same numbers in Figure 1.3)

(1) The C source program AA.c is compiled using nc100, then assembled using as100 to create the re-locatable object file AA.r30.
(2) The startup program ncrt0.a30 and the include file sect100.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 ln100, which is run from nc30, to create the absolute module file AA.x30.

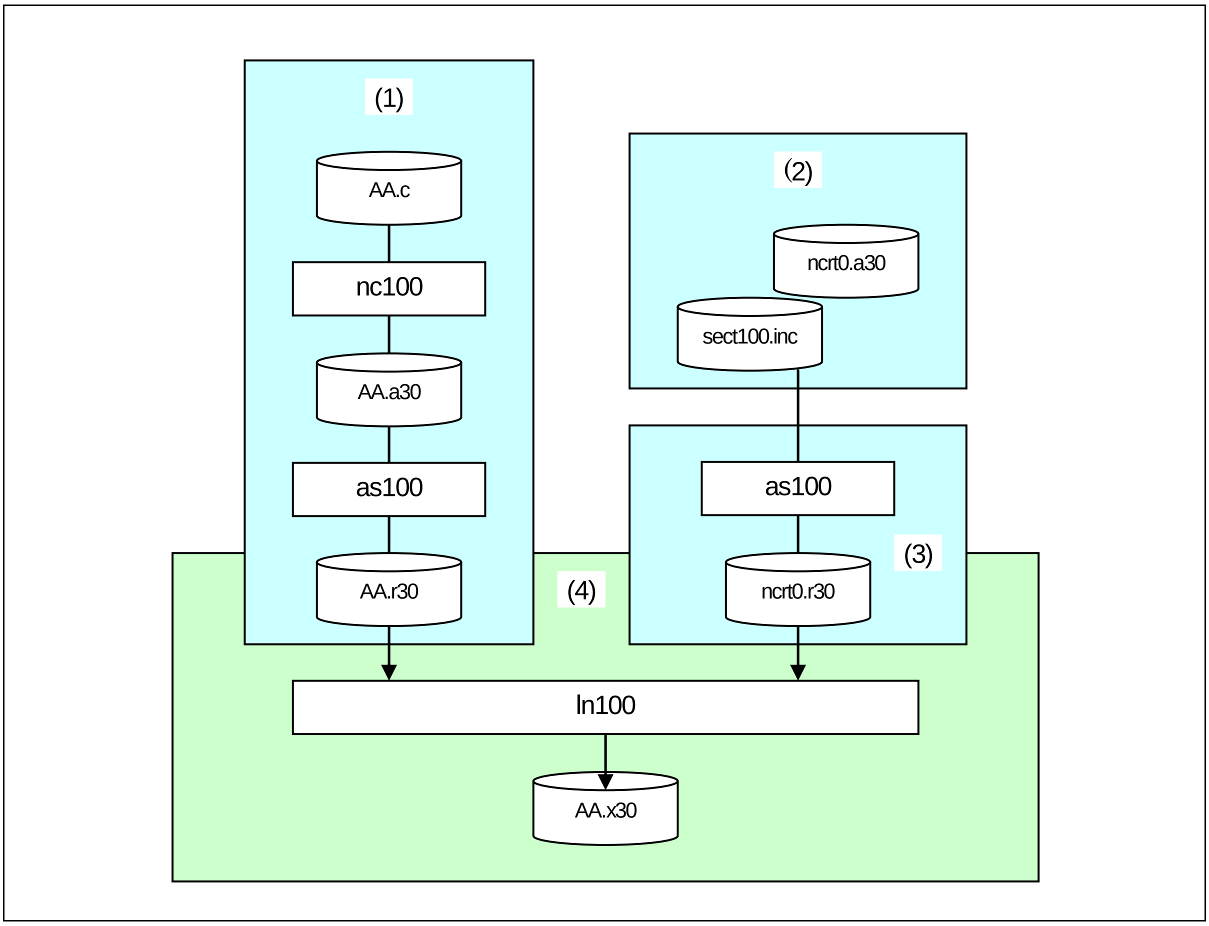


Figure 1.3 Program Development Flow

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

```
AA.x30 : ncrt0.r30 AA.r30
          nc100 -oAA ncrt0.r30 AA.r30

ncrt0.r30 : ncrt0.a30
          as100 ncrt0.a30

AA.r30 : AA.c
          nc100 -c AA.c
```

Figure 1.4 Example make File

Figure 1.5 shows the command line required for nc100 to perform the same operations as in the make file shown in Figure 1.4.

```
% nc100   -oAA   ncrt0.a30   AA.c<RET>

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

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

Figure 1.5 Example nc100 Command Line

1.5 NC100 Output Files

This chapter introduces the preprocess result C source program output when the sample program sample.c is compiled using NC100 and the assembly language source program.

1.5.1 Introduction to Output Files

With the specified command line options, the nc100 compile driver outputs the files shown in Figure 1.6. Below, we show the contents of the files output when the C source file smp.c shown in Figure 1.7 is compiled, assembled, and linked.
See the AS100 User Manual for the relocatable object files (extension .r30), print files (extension .lst),and map files (extension .map) output by as100 and ln100.

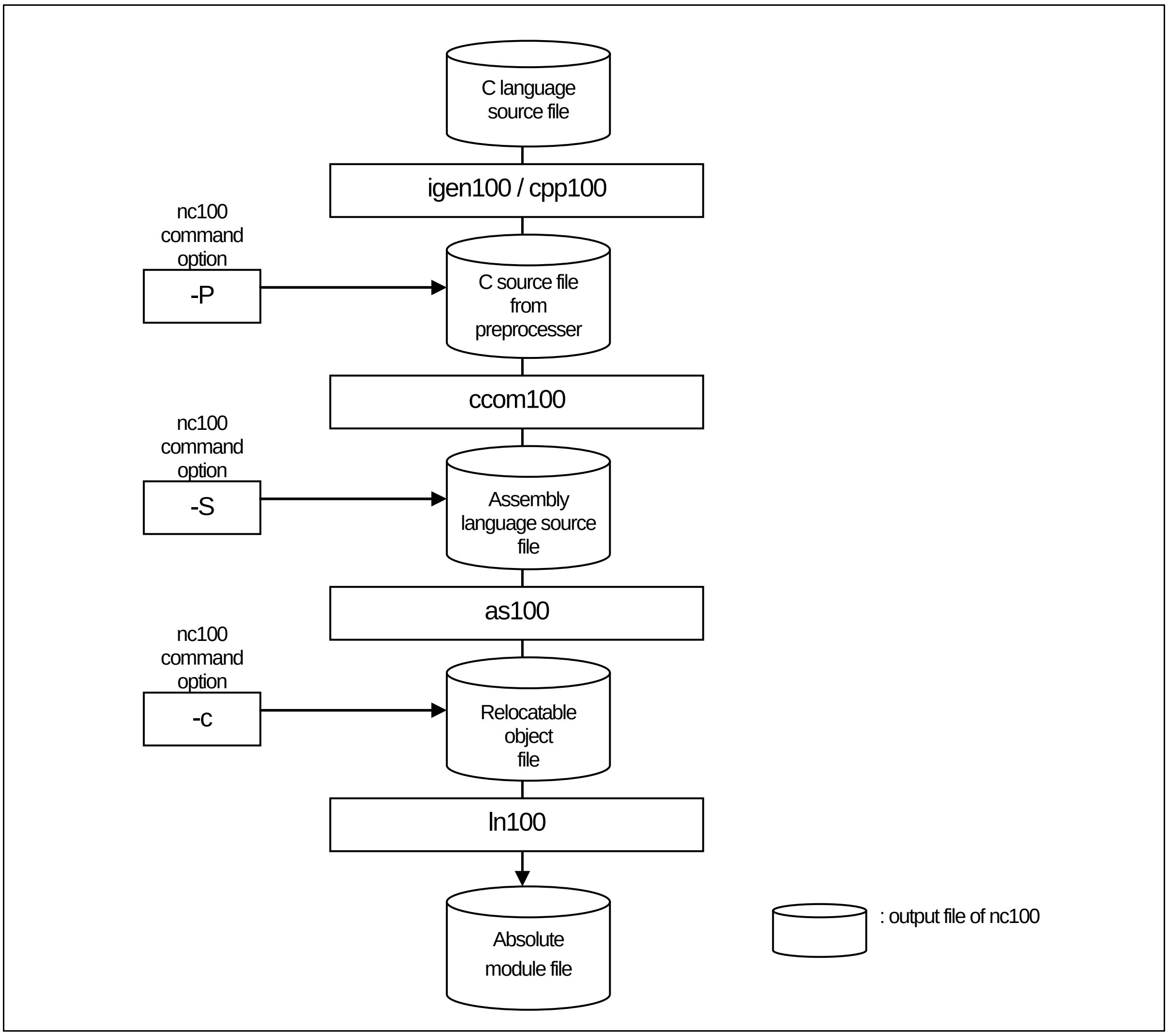


Figure 1.6 Relationship of nc100 Command Line Options and Output Files

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

void        main(void)
{
            int         flag;

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

Figure 1.7 Example C Source File (sample.c)

1.5.2 Preprocessed C Source Files

The cpp100 processes preprocess commands starting with #. Such operations include header file contents, macro expansion, and judgments on conditional compiling.
The C source files output by the preprocessor include the results of cpp100 processing of the C source files. Therefore, do not contain preprocess lines other than #pragma and #line. You can refer to these files to check the contents of programs processed by the compiler. The file extension is .i.
Figure 1.8 and Figure 1.9 are examples of file output.

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

typedef     unsigned long           size_t;

extern FILE _iob[];
```

Figure 1.8 Example Preprocessed C Source File (1/2)

```
typedef char _far          *__va_list;                                    (1)

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

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

             flag = 0 ;                          ← (3)

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

}
```

Figure 1.9   Example Preprocessed C Source File (2/2)

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

(1) Shows the expansion of header file stdio.h specified in #include.
(2) Shows the C source program resulting from expanding the macro.
(3) Shows that CLR specified in #define is expanded as 0.
(4) Shows that, because PRN specified in #define is 1, the compile condition is satisfied and the printf function is output.

1.5.3 Assembly Language Source Files

The assembly language source file is a file that can be processed by AS100 as a result of the compiler ccom100 converting the preprocess result C source file. The output files are assembly language source files with the extension .a30.
Figure 1.10 and Figure 1.11 are examples of the output files. When the nc100 command line option "-dsource (-dS) " is specified, the assembly language source files contain the contents of the C source file as comments.

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

;##      C Compiler                    OUTPUT
;## ccom100 Version X.XX.XX.XXX
;## Copyright(C) XXXX. Renesas Technology Corp.
;## and Renesas Solutions Corp., All Rights Reserved.
;## Compile Start Time XXX XX XX XX:XX:XX XXXX

;## COMMAND_LINE: ccom100    -dS -o sample.a30 sample.i

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

;## #         FUNCTION main
;## #         ARG Size(4)           Auto Size(0)          Context Size(4)


              .SECTION    program,CODE,ALIGN
              ._file      'sample.c'
              .align
              ._line      6
;## # C_SRC :             {
              .glb        _main
_main:
              ._line      9
;## # C_SRC :                         flag = CLR;
              mov.l       #00000000H,R2R0     ;    flag
              ._line      11
;## # C_SRC :                         printf("flag = %d¥n", flag);              <- (2)
              push.l      R2R0          ;    flag
              push.l      #___T0
              jsr         _printf
              add.l       #08H,SP
              ._line      13
;## # C_SRC :             }
              rts
E1:
                  :
              (omitted)
                  :
              .glb        __iob
              .glb        $getc
              .glb        _getchar
              .glb        $putc
              .glb        $putchar
              .glb        $feof
              .glb        $ferror
              .glb        $fgetc
              .glb        $fgets
              .glb        $fputc
                  :
              (omitted)
                  :
```

Figure 1.10 Example Assembly Language Source File (1/2) "sample.a30"

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

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

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

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

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

(2) When the nc100 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 nc100 and the command line options.

2.1 Starting Up the Compiler

2.1.1 nc100 Command Format

The nc100 compile driver starts the compiler commands (cpp100 and ccom100), the assemble command as100 and the link command ln100 to create a absolute module file. The following information (input parameters) is needed in order to start nc100:

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

These items are specified on the command line.

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

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

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

- Specifies machine language data file sample.x30. .... option -o
- Specifies output of list file (extension .lst) at assembling. .... option -as100 "-l"
- Specifies output of map file (extension .map) at linking .... option -ln100 "-ms"

```
% nc100△ [command-line-option]△[assembly-language-source-file-name]△
                    [relocatable-object-file-name]△<C-source-file-name>

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

Figure 2.1 nc100 Command Line Format

```
% nc100  -osample  -as100  "-l"  -ln100  "-ms"  ncrt0.a30  sample.c<RET>

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

Figure 2.2 Example nc100 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 Microsoft Windows (TM), etc.

a Command file input format

```
% nc100△[command-line-option]△&lt;@file-name&gt;[command-line-option]

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

Figure 2.3 Command File Command Line Format

```
% nc100   -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>
<CR>: Denotes carriage return.       -g   -as100   -l<CR>
                                     -o<CR>
                                     sample<CR>
```

Figure 2.5 Example Command File description

b Rules on command file description

The following rules apply for command file description:

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

c Precautions to be observed when using a command file

A directory path can be specified for command file names. An error results if the file does not exist in the specified directory path.
Command files for ln100 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 NC100 Command Line Options

a Notes on Coding nc100 Command Line Options

The nc100 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 opinion about control of 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 re-locatable file simultaneously with an assembly language source file, use the option "-dsource(shortcut -dS)".

2.1.4 nc100 Command Line Options

a Options for Controlling Compile Driver

Tabel 2.1 shows the command line options for controlling the compile driver.

Tabel 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.
-dsource (Short form -dS)	Generates an assembly language source file (extension ".a30") with a C language source list output as a comment. (Not deleted even after assembling.)
-dsource_in_list (Short form -dSL)	In addition to the "-dsource" function, generates an assembly language list file (.lst).
-E	Invokes only preprocess commands and outputs result to standard output.
-I*directory*	Specifies the directory containing the file(s) specified in #include. You can specify up to 50 directories.
-P	nvokes only preprocess commands and creates a file (extension .i).
-S	Creates an assembly language source file (extension .a30) and ends processing.
-silent	Suppresses the copyright message display at startup.
-U*predefined macro*	Undefines the specified predefined macro.

b Options Specifying Output Files

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

Tabel 2.2 Options for Specifying Output Files

Option	Function
-dir*directory-name*	Specifies the destination directory of the file(s) (absolute module file, map file, etc.) generated by ln100.
-o*file-name*	Specifies the name(s) of the file(s) (absolute module file, map file, etc.) generated by ln100. This option can also be used to specify the destination directory. Do not specify the filename extension.

c Version and command line Information Display Option

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

Tabel 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, nc100 finishes at ln100 and output files up to the absolute load module file (extension .x30) are created.

d Options for Debugging

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

Tabel 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.

e Optimization Options

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

Tabel 2.5 Optimization Option

Option	Short form	Function
-O[1-5]	None	Optimizes the program to be efficient in both speed and ROM size at each level.
-O5OA	None	Inhibits code generation based on bit-manipulating instructions when the optimization option "-O5" is selected.
-OR	None	Optimizes the program as much as possible by placing priority on ROM size.
-OS	None	Optimizes the program as much as possible by placing priority on speed.
-OR_MAX	-ORM	Maximum optimization of ROM size followed by speed.
-OS_MAX	-OSM	Maximum optimization of speed followed by ROM size.
-Ocompare_byte_to_word	-OCBTW	Compares consecutive bytes of data at contiguous addresses in words.
-Oconst	-OC	Performs optimization by replacing references to the const-qualified external variables with constants.
-Ofile_inline	-OFI	All inline functions are expanded inline.
-Oinline_line	-OIL	This option changes the size (number of lines) of the function to be inline expanded.
-Oglb_jmp	-OGJ	Global jump is optimized.
-Oglobal_to_inline	-OGTI	Handles global functions as inline-declared.
-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_bit	-ONB	Suppresses optimization based on grouping of bit manipulations.
-Ono_break_source_debug	-ONBSD	Suppresses optimization that affects source line data.
-Ono_float_const_fold	-ONFCF	Suppresses the constant folding processing of floating point numbers.
-Ono_logical_or_combine	-ONLOC	Suppresses the optimization that puts consecutive OR together.
-Ono_asmopt	-ONA	Inhibits starting the assembler optimizer "aopt100".
-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.
-Ostatic_to_inline	-OSTI	A static function is treated as an inline function.

f Generated Code Modification Options

Tabel 2.6 shows the command line options for controlling nc100 generated assembly code.

Tabel 2.6 Generated Code Modification Options

Option	Short form	Function
-fansi	None	Makes "-fnot_reserve_far_and_near", "-fnot_reserve_asm", and "-fextend_to_int" valid.
-fconst_not_ROM	-fCNR	Does not handle the types specified by const as ROM data.
-fdouble_32	-fD32	This option specifies that the double type be handled in 32-bit data length as is the float type.
-fenable_register	-fER	Make register storage class available.
-fextend_to_int	-fETI	Performs operation after extending char-type data to the int type. (Extended according to ANSI standards.)[1]
-ffar_RAM	-fFRAM	Changes the default attribute of RAM data to far.
-finfo	None	Outputs the information required for the Inspector, "Call Walker", and "Map Viewer" to the absolute module file (.x30).
-fint_16	-fI16	Does handle int type at the 16-bit width.
-fJSRW	None	Changes the default instruction for calling functions to JSR.W.
-fnear_ROM	-fNROM	Changes the default attribute of ROM data to near.
-fno_align	-fNA	Does not align the start address of the function.
-fno_switch_table	-fNST	When this option is specified, the code which branches since it compares is generated to a switch statement.
-fnot_address_volatile	-fNAV	Does not regard the variables specified by #pragma ADDRESS (#pragma EQU) as those specified by volatile.
-fnot_reserve_asm	-fNRA	Exclude asm from reserved words. (Only _asm is valid.)
-fnot_reserve_far_and_near	-fNRFAN	Exclude far and near from reserved words. (Only _far and _near are valid.)
-fnot_reserve_inline	-fNRI	Exclude far and near from reserved words. (Only _inline is made a reserved word.)
-fsigned_char	-fSC	Handles type char without sign specification as type signed char.
-fswitch_other_section	-fSOS	This option outputs a ROM table for a 'switch' statement to some other section than a program section.
-fuse_FPU	-fUF	Outputs FPU instruction

[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.

g Library Specifying Option

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

Tabel 2.7 Library Specifying Option

Option	Function
-l*libraryfilename*	Specifies a library file that is used by ln308 when linking files.

h Warning Options

Tabel 2.8 shows the command line options for outputting warning messages for contraventions of nc100 language specifications.

Tabel 2.8 Warning Options

Option	Short form	Function
-Wall	None	Displays message for all detectable warnings. (however, not including alarms output by -Wlarge_to_small and "-Wno_used_argument")
-Wccom_max_warnings =*Warning Count*	-WCMW	This option allows you to specify an upper limit for the number of warnings output by ccom100.
-Werror_file<*file name*>	-WEF	Outputs error messages to the specified file.
-Wlarge_to_small	-WLTS	Outputs a warning about the tacit transfer of variables in descending sequence of size.
-Wmake_tagfile	-WMT	Outputs error messages to the tag file of source file by source file.
-Wnesting_comment	-WNC	Outputs a warning for a comment including "*/" .
-Wno_stop	-WNS	Prevents the compiler stopping when an error occurs.
-Wno_used_argument	-WNUA	Outputs a warning for unused argument of functions.
-Wno_used_function	-WNUF	Displays unused global functions when linking.
-Wno_used_static_function	-WNUSF	For one of the following reasons, a static function name is output that does not require code generation.
-Wno_warning_stdlib	-WNWS	Specifying this option while "-Wnon_prototype" or "-Wall" is specified inhibits "Alarm for standard libraries which do not have prototype declaration.
-Wnon_prototype	-WNP	Outputs warning messages for functions without prototype declarations.
-Wstdout	None	Outputs error messages to the host machine's standard output (stdout).
-Wstop_at_link	-WSAL	Stops linking the source files if a warning occurs during linking to suppress generation of absolute module files. Also, a return value "10" is returned to the host OS.
-Wstop_at_warning	-WSAW	Stops compiling the source files if a warning occurs during compiling and returns the compiler end code "10".
-Wundefined_macro	-WUM	Warns you that undefined macros are used in #if.
-Wuninitialize_variable	-WUV	Outputs a warning about auto variables that have not been initialized.
-Wunknown_pragma	-WUP	Outputs warning messages for non-supported #pragma.
-Wmultiple_tentative_definitions	-WMTD	Outputs a warning when there are multiple tentative definitions for one and the same variable name.
-Wignore_near_pointer	-WINP	Inhibits a warning when the near pointer is handled as a far pointer.

i Assemble and Link Options

Tabel 2.9 shows the command line options for specifying as100 and ln100 options.

Tabel 2.9 Assemble and Link Options

Option	Function
-as100△< Option>	Specifies options for the as100 link command. If you specify two or more options, enclose them in double quotes.
-ln100△< Option>	Specifies options for the ln100 assemble command. If you specify two or more options, enclose them in double quotes.

2.2 Preparing the Startup Program

For C-language programs to be "burned" into ROM, NC100 comes with a sample startup program written in the assembly language to initial set the hardware (R32C/100), 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 NC100 startup program consists of the following two files:

- ncrt0.a30
  Write a program which is executed immediately after reset.
- sect100.inc
  Included from ncrt0.a30, this file defines section locations (memory mapping).

Figure 2.6 to Figure 2.11 show the ncrt0.a30 source program list. Figure 2.12 to Figure 2.17 show the sect100.inc source program list.

```
;************************************************************************;
;
;   C COMPILER for R32C/100
;   Copyright(C) XXXX. Renesas Technology Corp.
;   and Renesas Solutions Corp., All rights reserved.
;
;   ncrt0.a30 : startup program
;
;   This program is applicable when using the basic I/O library
;
;   $Id: ncrt0.a30,v X.XX XXXX/XX/XX XX:XX:XX XXXXX Exp $
;
;************************************************************************;

;---------------------------------------------------------------------
; HEEP SIZE definition                                          ← (1)
;---------------------------------------------------------------------
.if __HEAP__ == 1                       ; for HEW

HEAPSIZE .equ          0h

.else
.if __HEAPSIZE__ == 0

HEAPSIZE .equ          300h

.else                                           ; for HEW

HEAPSIZE .equ          __HEAPSIZE__

.endif
.endif;

 (1) defines the heap size.
```

Figure 2.6 Startup Program List (1/12) (ncrt0.a30)

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

STACKSIZE          .equ       300h

.else                                    ; for HEW

STACKSIZE          .equ       __USTACKSIZE__

.endif

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

ISTACKSIZE         .equ       300h

.else                                    ; for HEW

ISTACKSIZE         .equ       __ISTACKSIZE__

.endif

;---------------------------------------------------------------------
; INTERRUPT VECTOR ADDRESS    definition                         ← (4)
;---------------------------------------------------------------------
VECTOR_ADR         .equ       0FFFFFBDCH

;---------------------------------------------------------------------
; Section allocation
;---------------------------------------------------------------------
          .list OFF
          .include sect100.inc                                   ← (5)
          .list ON
```

(2) defines the user stack size.
(3) defines the interrupt stack size.
(4) defines the start address of interrupt vector table.
(5) Includes sect100.inc

Figure 2.7 Startup Program List (2/12) (ncrt0.a30)

```
;---------------------------------------------------------------------;
; SB AREA DEFINITION                                                  ;
;---------------------------------------------------------------------;
                .glb        __SB__
__SB__          .equ        data_SB8_top
;---------------------------------------------------------------------;
; INITIALIZE MACRO DEFINITION                                         ;
;---------------------------------------------------------------------;
BZERO           .macro      TOP_,SECT_
                mov.b       #00H,R0L
                mov.l       #TOP_,A1
                mov.l       #sizeof SECT_,R7R5
                sstr.b
                .endm
BCOPY           .macro      FROM_,TO_,SECT_
                mov.l       #FROM_,A0
                mov.l       #TO_,A1
                mov.l       #sizeof SECT_,R7R5
                smovf.b
                .endm
```

Figure 2.8 Startup Program List (3/12) (ncrt0.a30)

```
;---------------------------------------------------------------------;
; INTERRUPT SECTION                                                    ;
;---------------------------------------------------------------------;
                .insf           start, S, 0
                .glb            start
                .section        interrupt, code, align
start:                                                          ← (6)
                ;-----------------------------------------------;
                ; after reset, this program will start          ;
                ;-----------------------------------------------;
                ldc             #istack_top,ISP     ; istack pointer
                ldc             #0080H,FLG          ; switch to usp     ← (7)
                ldc             #stack_top,SP       ; stack pointer
                ldc             #data_SB8_top,SB    ; sb register
                fset            b
                ldc             #data_SB8_top,SB    ; bsb register
                fclr            b
                ldc             #VECTOR_ADR,INTB ; vector address
                ;-----------------------------------------------;
                ; zero clear BSS                                ;  ← (8)
                ;-----------------------------------------------;
                BZERO           bss_SB8_top,    bss_SB8
;               BZERO           bss_SB16_top, bss_SB16
                BZERO           bss_NEAR_top, bss_NEAR
                BZERO           bss_FAR_top,    bss_FAR
                BZERO           bss_EXT_top,    bss_EXT
                BZERO           bss_MON1_top, bss_MON1
                BZERO           bss_MON2_top, bss_MON2
                BZERO           bss_MON3_top, bss_MON3
                BZERO           bss_MON4_top, bss_MON4
                ;-----------------------------------------------;
                ; initialize DATA                               ;  ← (9)
                ;-----------------------------------------------;
                BCOPY           data_SB8_INIT_top,    data_SB8_top,    data_SB8
;               BCOPY           data_SB16_INIT_top, data_SB16_top, data_SB16
                BCOPY           data_NEAR_INIT_top, data_NEAR_top, data_NEAR
                BCOPY           data_FAR_INIT_top,    data_FAR_top,    data_FAR
                BCOPY           data_EXT_INIT_top,    data_EXT_top,    data_EXT
                BCOPY           data_MON1_INIT_top, data_MON1_top, data_MON1
                BCOPY           data_MON2_INIT_top, data_MON2_top, data_MON2
                BCOPY           data_MON3_INIT_top, data_MON3_top, data_MON3
                BCOPY           data_MON4_INIT_top, data_MON4_top, data_MON4
```

(6) After a reset, execution starts from this label (start)
(7) Sets IPL and each flags.
(8) Clears the bss section (to zeros).
(9) Moves the initial values of the data section to RAM.

Figure 2.9 Startup Program List (4/12) (ncrt0.a30)

```
                        ;---------------------------------------------------;
                        ; initialize heap manager                           ;         ← (10)
                        ;---------------------------------------------------;
.if __HEAP__ != 1
                         .glb          __mnext
                         .glb          __msize
                         mov.l         #heap_top,__mnext
                         mov.l         #HEAPSIZE,__msize
.endif
                        ;---------------------------------------------------;
                        ; initialize standard I/O                           ;         ← (11)
                        ;---------------------------------------------------;
.if __STANDARD_IO__ == 1
                         .glb          __init
                         .call         __init, G
                         jsr.a         __init
.endif
                        ;---------------------------------------------------;
                        ; invoke main() function                            ;         ← (12)
                        ;---------------------------------------------------;
                        ldc            #0H,FB      ; for DEBUGGER
                        .glb           _main
                        jsr.a          _main
```

(10) Initializes the heap area. Comment out this line if no memory management function is used.
(11) Calls the init function, which initializes standard I/O. Comment out this line if no I/O function is used.
(12) Calls the 'main' function.
* Interrupt is not enable, when calls 'main' function.
  Therefore, permits interrupt by FSET command, when uses interrupt function.

Figure 2.10 Startup Program List (5/12) (ncrt0.a30)

```
;======================================================================
                ;-------------------------------------------------------;
                ; exit() function                                       ;        ← (13)
                ;-------------------------------------------------------;
                .glb            _exit
                .glb            $exit
_exit:                                          ; End of execution
$exit:
                jmp             _exit
                .einsf
                ;-------------------------------------------------------;
                ; dummy interrupt function                              ;        ← (14)
                ;-------------------------------------------------------;
                .glb            dummy_int
dummy_int:
                reit
                .end
;*************************************************************************;
;                                                                         ;
;          End of R32C/100 start up                                       ;
;                                                                         ;
;*************************************************************************;
```

(13) exit function.
(14) Dummy interrupt processing function.

Figure 2.11 Startup Program List (6/12) (ncrt0.a30)

```
;*************************************************************************;
;
;   C COMPILER for R32C/100
;   Copyright(C) XXXX. Renesas Technology Corp.
;   and Renesas Solutions Corp., All rights reserved.
;
;   ncrt0.a30 : startup program
;
;   This program is applicable when using the basic I/O library
;
;   $Id: sect100.inc,v X.X XXXX/XX/XX XX:XX:XX XXX Exp $
;
;*************************************************************************;
;---------------------------------------------------------------------;
;
;               Arrangement of section
;
;---------------------------------------------------------------------;
;---------------------------------------------------------------------;
; NEAR RAM SECTIONS                                                   ;
;---------------------------------------------------------------------;
                .section    data_SB8,   data
                .org                    00000400H
data_SB8_top:
                .section    bss_SB8,                data, align
bss_SB8_top:
                .section    data_NEAR,              data, align
data_NEAR_top:
                .section    bss_NEAR,               data, align
bss_NEAR_top:
                .section    data_MON1,              data, align
data_MON1_top:
                .section    bss_MON1,               data, align
bss_MON1_top:
                .section    data_MON2,              data, align
data_MON2_top:
                .section    bss_MON2,               data, align
bss_MON2_top:
                .section    data_MON3,              data, align
data_MON3_top:
                .section    bss_MON3,               data, align
bss_MON3_top:
                .section    data_MON4,              data, align
data_MON4_top:
                .section    bss_MON4,               data, align
bss_MON4_top:

;---------------------------------------------------------------------;
; STACK SECTION                                                       ;
;---------------------------------------------------------------------;
                .section    stack,                  data, align
                .blkb                   STACKSIZE
                .align
stack_top:
                .blkb                   ISTACKSIZE
                .align
istack_top:
```

Figure 2.12 Startup Program List (7/12) (sect100.inc)

```
;---------------------------------------------------------------;
; HEAP SECTION                                                  ;
;---------------------------------------------------------------;
                .section    heap,       data, align
heap_top:
                .blkb                   HEAPSIZE

;---------------------------------------------------------------;
; SB RELATIVE RAM SECTIONS                                      ;
;---------------------------------------------------------------;
;               .section    data_SB8,   data
;               .org                    00008000H
;data_SB8_top:
;               .section    bss_SB8,    data, align
;bss_SB8_top:
;               .section    data_SB16,data, align
;data_SB16_top:
;               .section    bss_SB16,   data, align
;bss_SB16_top:

;---------------------------------------------------------------;
; FAR RAM SECTIONS                                              ;
;---------------------------------------------------------------;
                .section    data_FAR,   data, align
data_FAR_top:
                .section    bss_FAR,    data, align
bss_FAR_top:

;---------------------------------------------------------------;
; EXTENDED RAM SECTIONS                                         ;
;---------------------------------------------------------------;
                .section    data_EXT,   data
                .org                    00800000H
data_EXT_top:
                .section    bss_EXT,    data, align
bss_EXT_top:

;---------------------------------------------------------------;
; EXTENDED ROM SECTIONS                                         ;
;---------------------------------------------------------------;
                .section    data_EXT_INIT,          romdata
                .org                    0FF000000H
data_EXT_INIT_top:
                .section    rom_EXT,    romdata, align
rom_EXT_top:
                .section    program_EXT,            code, align

;---------------------------------------------------------------;
; FAR ROM SECTIONS                                              ;
;---------------------------------------------------------------;
                .section    rom_FAR,    romdata
                .org                    0FFE00000H
rom_FAR_top:
```

Figure 2.13 Startup Program List (8/12) (sect100.inc)

```
;---------------------------------------------------------------;
; INITIAL DATA SECTIONS                                         ;
;---------------------------------------------------------------;
                        .section    data_NEAR_INIT,     romdata
                        .org        0FFFF0000H
data_NEAR_INIT_top:
                        .section    data_MON1_INIT,     romdata, align
data_MON1_INIT_top:
                        .section    data_MON2_INIT,     romdata, align
data_MON2_INIT_top:
                        .section    data_MON3_INIT,     romdata, align
data_MON3_INIT_top:
                        .section    data_MON4_INIT,     romdata, align
data_MON4_INIT_top:
                        .section    data_SB8_INIT,      romdata, align
data_SB8_INIT_top:
;                       .section    data_SB16_INIT,     romdata, align
;data_SB16_INIT_top:
                        .section    data_FAR_INIT,      romdata, align
data_FAR_INIT_top:

;---------------------------------------------------------------;
; SWITCH TABLE SECTIONS                                         ;
;---------------------------------------------------------------;
                        .section    switch_table,       romdata, align

;---------------------------------------------------------------;
; CODE SECTIONS                                                 ;
;---------------------------------------------------------------;
                        .section    program,    code, align

                        .section    interrupt,  code, align

;---------------------------------------------------------------;
; NEAR ROM SECTIONS                                             ;
;---------------------------------------------------------------;
;                       .section    rom_NEAR,           romdata
;                       .org                    0FFFF8000H
;rom_NEAR_top:
```

Figure 2.14 Startup Program List (9/12) (sect100.inc)

```
;---------------------------------------------------------------;
; VARIABLE VECTOR SECTION                                       ;
;---------------------------------------------------------------;
                .section    vector,     romdata
                .org        VECTOR_ADR
.if     __MVT__ == 1
        .lword  dummy_int           ; BRK                   (software int 0)
        .lword  dummy_int           ; reservation area      (software int 1)
        .lword  dummy_int           ; uart5 trance/NACK     (software int 2)
        .lword  dummy_int           ; uart5 receive/ACK     (software int 3)
        .lword  dummy_int           ; uart6 trance/NACK     (software int 4)
        .lword  dummy_int           ; uart6 receive/ACK     (software int 5)
        .lword  dummy_int           ; uart5/uart6 bus collision (software int 6)
        .lword  dummy_int           ; reservation area      (software int 7)
        .lword  dummy_int           ; DMA0                  (software int 8)
        .lword  dummy_int           ; DMA1                  (software int 9)
        .lword  dummy_int           ; DMA2                  (software int 10)
        .lword  dummy_int           ; DMA3                  (software int 11)
        .lword  dummy_int           ; TIMER A0              (software int 12)
        .lword  dummy_int           ; TIMER A1              (software int 13)
        .lword  dummy_int           ; TIMER A2              (software int 14)
        .lword  dummy_int           ; TIMER A3              (software int 15)
        .lword  dummy_int           ; TIMER A4              (software int 16)
        .lword  dummy_int           ; uart0 trance/NACK     (software int 17)
        .lword  dummy_int           ; uart0 receive/ACK     (software int 18)
        .lword  dummy_int           ; uart1 trance/NACK     (software int 19)
        .lword  dummy_int           ; uart1 receive/ACK     (software int 20)
        .lword  dummy_int           ; TIMER B0              (software int 21)
        .lword  dummy_int           ; TIMER B1              (software int 22)
        .lword  dummy_int           ; TIMER B2              (software int 23)
        .lword  dummy_int           ; TIMER B3              (software int 24)
        .lword  dummy_int           ; TIMER B4              (software int 25)
        .lword  dummy_int           ; INT5                  (software int 26)
        .lword  dummy_int           ; INT4                  (software int 27)
        .lword  dummy_int           ; INT3                  (software int 28)
        .lword  dummy_int           ; INT2                  (software int 29)
        .lword  dummy_int           ; INT1                  (software int 30)
        .lword  dummy_int           ; INT0                  (software int 31)
        .lword  dummy_int           ; TIMER B5              (software int 32)
        .lword  dummy_int           ; uart2 trance/NACK     (software int 33)
        .lword  dummy_int           ; uart2 receive/ACK     (software int 34)
        .lword  dummy_int           ; uart3 trance/NACK     (software int 35)
        .lword  dummy_int           ; uart3 receive/ACK     (software int 36)
        .lword  dummy_int           ; uart4 trance/NACK     (software int 37)
        .lword  dummy_int           ; uart4 receive/ACK     (software int 38)
        .lword  dummy_int           ; uart2       bus collision (software int 39)
        .lword  dummy_int           ; uart3/uart0 bus collision (software int 40)
        .lword  dummy_int           ; uart4/uart1 bus collision (software int 41)
        .lword  dummy_int           ; A-D Convert           (software int 42)
        .lword  dummy_int           ; input key             (software int 43)
        .lword  dummy_int           ; intelligent I/O  0    (software int 44)
        .lword  dummy_int           ; intelligent I/O  1    (software int 45)
        .lword  dummy_int           ; intelligent I/O  2    (software int 46)
        .lword  dummy_int           ; intelligent I/O  3    (software int 47)
        .lword  dummy_int           ; intelligent I/O  4    (software int 48)
        .lword  dummy_int           ; intelligent I/O  5    (software int 49)
        .lword  dummy_int           ; intelligent I/O  6    (software int 50)
        .lword  dummy_int           ; intelligent I/O  7    (software int 51)
        .lword  dummy_int           ; intelligent I/O  8    (software int 52)
        .lword  dummy_int           ; intelligent I/O  9    (software int 53)
        .lword  dummy_int           ; intelligent I/O 10    (software int 54)
```

Figure 2.15 Startup Program List (10/12) (sect100.inc)

```
.lword    dummy_int    ; intelligent I/O 11     (software int 55)
.lword    dummy_int    ; reservation area       (software int 56)
.lword    dummy_int    ; reservation area       (software int 57)
.lword    dummy_int    ; reservation area       (software int 58)
.lword    dummy_int    ; CAN1WU                 (software int 59)
.lword    dummy_int    ; reservation area       (software int 60)
.lword    dummy_int    ; reservation area       (software int 61)
.lword    dummy_int    ; reservation area       (software int 62)
.lword    dummy_int    ; reservation area       (software int 63)
.lword    dummy_int    ; reservation area       (software int 64)
.lword    dummy_int    ; reservation area       (software int 65)
.lword    dummy_int    ; reservation area       (software int 66)
.lword    dummy_int    ; reservation area       (software int 67)
.lword    dummy_int    ; Audio interface 0      (software int 68)
.lword    dummy_int    ; Sound field processor (software int 69)
.lword    dummy_int    ; reservation area       (software int 70)
.lword    dummy_int    ; reservation area       (software int 71)
.lword    dummy_int    ; reservation area       (software int 72)
.lword    dummy_int    ; reservation area       (software int 73)
            :
          (omitted)
            :
.lword    dummy_int    ; reservation area       (software int 89)
.lword    dummy_int    ; reservation area       (software int 90)
.lword    dummy_int    ; reservation area       (software int 91)
.lword    dummy_int    ; reservation area       (software int 92)
.lword    dummy_int    ; INT8                   (software int 93)
.lword    dummy_int    ; INT7                   (software int 94)
.lword    dummy_int    ; INT6                   (software int 95)
.lword    dummy_int    ; CAN0 trance            (software int 96)
.lword    dummy_int    ; CAN0 receive           (software int 97)
.lword    dummy_int    ; CAN0 error             (software int 98)
.lword    dummy_int    ; CAN1 trance            (software int 99)
.lword    dummy_int    ; CAN1 receive           (software int 100)
.lword    dummy_int    ; CAN1 error             (software int 101)
.lword    dummy_int    ; reservation area       (software int 102)
.lword    dummy_int    ; reservation area       (software int 103)
.lword    dummy_int    ; reservation area       (software int 104)
.lword    dummy_int    ; reservation area       (software int 105)
            :
          (omitted)
            :
.lword    dummy_int    ; reservation area       (software int 120)
.lword    dummy_int    ; reservation area       (software int 121)
.lword    dummy_int    ; reservation area       (software int 122)
.lword    dummy_int    ; reservation area       (software int 123)
.lword    dummy_int    ; uart7 trance           (software int 124)
.lword    dummy_int    ; uart7 receive          (software int 125)
.lword    dummy_int    ; uart8 trance           (software int 126)
.lword    dummy_int    ; uart8 receive          (software int 127)
.lword    dummy_int    ; software int 128
.lword    dummy_int    ; software int 129
.lword    dummy_int    ; software int 130
.lword    dummy_int    ; software int 131
.lword    dummy_int    ; software int 132
```

Figure 2.16   Startup Program List (11/12) (sect100.inc)

```
                .lword      dummy_int               ; software int 133
                .lword      dummy_int               ; software int 134
                .lword      dummy_int               ; software int 135
                               :
                            (omitted)
                               :
                .lword      dummy_int               ; software int 253
                .lword      dummy_int               ; software int 254
                .lword      dummy_int               ; software int 255
.endif
.
;---------------------------------------------------------------;
; FIXED VECTOR SECTION                                          ;
;---------------------------------------------------------------;
                            .section    fvector,    romdata
                            .org                    0FFFFFFDCH
UDI:                        .lword                  dummy_int
OVER_FLOW:                  .lword                  dummy_int
BRKI:                       .lword                  dummy_int
                            .lword                  0FFFFFFFFH
                            .lword                  0FFFFFFFFH
WDT:                        .lword                  dummy_int
                            .lword                  dummy_int
NMI:                        .lword                  dummy_int
RESET:                      .lword                  start

;---------------------------------------------------------------;
; ID code    DEFINITION ;
;---------------------------------------------------------------;
; ID code check function
; .id "CodeChk"

;***************************************************************************;
;                                                                           ;
;           End of R32C/100 start up                                        ;
;                                                                           ;
;***************************************************************************;
```

Figure 2.17 Startup Program List (12/12) (sect100.inc)

2.2.2 Customizing the Startup Program

a Overview of Startup Program Processing

(1) About ncrt0.a30

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

- Sets the top address (__SB__) of the SBDATA area (it is accessing area to used the SB relative addressing mode).
- Sets the processor's operating mode.
- Initializes the stack pointer (ISP Register and USP Register).
- Initializes SB register.
- Initializes INTB register.
- Initializes the data near area.
  (1) Default
  bss_NEAR sections are cleared (to 0).
  (2) When far-qualified variables are used
  bss_FAR sections are cleared (to 0).
  (3) #pragma SBDATA Extended Functions
  bss_SB8 sections are cleared (to 0).
  (4) #pragma SB16DATA Extended Functions
  bss_SB16 sections are cleared (to 0).
  (5) #pragma EXTMEM Extended Functions
  bss_EXT sections are cleared (to 0).
  (6) #pragma MONITORn Extended Functions
  bss_MON1, bss_MON2, bss_MON3, bss_MON4 sections are cleared (to 0).
- Transfers initial values from the ROM section in which they are stored to a data area that has initial values.
  (1) Default
  Transfers initial values from the data_NEAR_INIT section to the data_NEAR section.
  (2) When far-qualified variables are used
  Transfers initial values from the data_FAR_INIT section to the data_FAR section.
  (3) #pragma SBDATA Extended Functions
  Transfers initial values from the data_SB8_INIT section to the data_SB8 section.
  (4) #pragma SB16DATA Extended Functions
  Transfers initial values from the data_ SB16_INIT section to the data_SB16 section.
  (5) #pragma EXTMEM Extended Functions
  Transfers initial values from the data_EXT_INIT section to the data_EXT section.
  (6) #pragma MONITORn Extended Functions
  Transfers initial values from the data_MON1_INIT, data_MON2_INIT, data_MON3_INIT, and data_MON4_INIT sections to data_MON1, data_MON2, data_MON3, and data_MON4 sections, respectively.
  - 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.

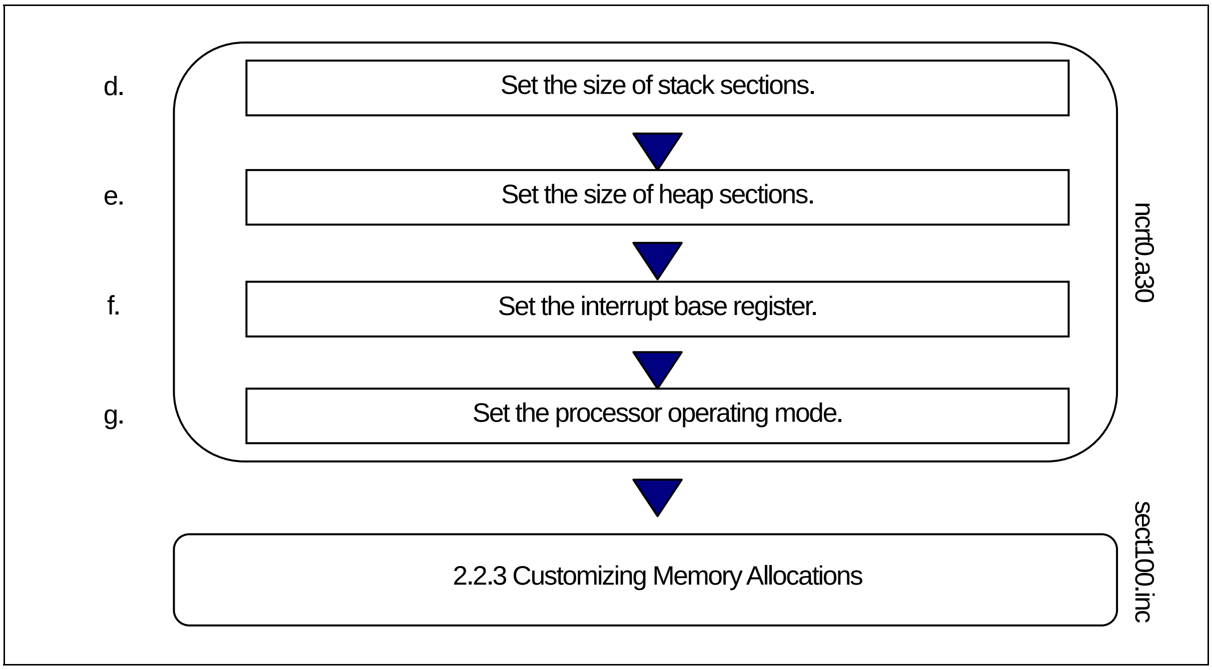


Figure 2.18 Example Sequence for Modifying Startup Programs

c Examples of startup modifications that require caution

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

The init function[1] initializes the R32C/100 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                       ;
                            ;-----------------------------------------------;
    .if __STANDARD_IO__ == 1
                            .glb            __init
                            .call           __init,G
                            jsr.a           __init
    .endif
```

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 R32C/100 is assumed to be the microcomputer that it initializes.
When using standard input/output functions, the init function, etc. may need to be modified depending on the system in which the microcomputer is to be used.

(2) Settings When Not Using Memory Management Functions

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

(1) Initialization of external variable char *_mnext

Initializes the heap_top label, which is the starting address of the heap section.

(2) Initialization of external variable unsigned_msize

Initializes the "HEAPSIZE" expression, which sets at "2.2.2 e heap section size".

Figure 2.20 shows the initialization performed in ncrt0.a30.

```
                    ;----------------------------------------------------;
                    ; initialize heap manager                            ;
                    ;----------------------------------------------------;
.if __HEAP__  != 1
                    .glb            __mnext
                    .glb            __msize
                    mov.l           #heap_top,__mnext
                    mov.l           #HEAPSIZE,__msize
.endif
```

Figure 2.20 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:

- Call the C subroutine only after clearing them, B and D flags.
- Call the C subroutine only after setting the U flag.

d Setting the Stack Section Size

A stack section has the domain used for user stacks, and the domain used for interruption stacks. Since stack is surely used, please surely secure a domain. stack size should set up the greatest size to be used.[1]
Stack size is calculated to use the stack size calculation utility Call Walker.

[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.

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.

```
;---------------------------------------------------------------
; HEEP SIZE definition
;---------------------------------------------------------------
.if __HEAP__ == 1                               ; for HEW

HEAPSIZE .equ       0h

.else
.if __HEAPSIZE__ == 0

HEAPSIZE .equ       300h

.else                                           ; for HEW

HEAPSIZE .equ       __HEAPSIZE__

.endif
.endif
```

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
;---------------------------------------------------------------
VECTOR_ADR      .equ      0FFFFFBDCH
```

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.

0FFFFFBDCH - 0FFFFFFDBH:	Interrupt vector table
0FFFFFFDCH - 0FFFFFFFFH:	Fixed vector table

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

g Setting the Processor Mode Register

Set the processor operating mode to match the target system at address 04H (Processor mode register) in the part of ncrt0.a30 shown in Figure 2.23.

```
;-------------------------------------------------------;
; after reset, this program will start                  ;
;-------------------------------------------------------;
    :
(omitted)
    :
; mov.b            #00H,04H            ; processor 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 NC100 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 NC100, the user must determine the memory mapping.
With NC100, storage class variables, variables with initial values, variables without initial values, character string data, interrupt processing programs, and interrupt vector address tables, etc., are mapped to Micoro Processor series memory as independent sections according to their function.
The names of sections consist of a base name and attribute as shown below:

```
[ Section Base Name ] ___ [ Attri ]
```

Figure 2.24 Section Names

Tabel 2.10 shows Section Base Name and Tabel 2.11 shows Attributes.

Tabel 2.10 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

Tabel 2.11 Section Naming Rules

Attribute	Meaning		Target section base name
INIT	Section containing initial values of data		data
NEAR / FAR / SB8 / SB16 / EXT / MON1 / MON2 / MON3 / MON4	NEAR	near attribute [1]	data, bss, rom
	FAR	far attribute	
	SB8	SBDATA attribute	data, bss
	SB16	SB16DATA attribute	data, bss
	EXT	EXTMEM attribute	data, bss, rom
	MON1	MONITOR1 attribute	data, bss
	MON2	MONITOR2 attribute	data, bss
	MON3	MONITOR3 attribute	data, bss
	MON4	MONITOR4 attribute	data, bss

Tabel 2.12 shows the contents of sections other than those based on the naming rules described above.

Tabel 2.12 Section Names

Section name	Contents
fvector	This section stores the contents of the Micro Processor's fixed vector.
heap	This memory area is dynamically allocated during program execution by memory management functions (e.g., malloc). This section can be allocated at any desired location of the Micro Processor RAM area.
program	Stores programs
stack	This area is used as a stack. Allocate this area at addresses between 0400H to 7FFFH.
switch_table	The section to which the branch table for switch statements is allocated. This section is generated only with the "-fSOS" option.
vector	This section stores the contents of the Micro Processor's interrupt vector table. The interrupt vector table can be allocated at any desired location of the Micro Processor's entire memory space by intb register relative addressing. For more information, refer to the Micro Processor User's Manual.

These sections are mapped to memory according to the settings in the startup program include file sect100.inc. You can modify the include file to change the mapping.
Figure 2.25 shows the how the sections are mapped according to the sample startup program's include file sect100.inc.

[1] near and far are the qualifiers specific to NC100. Use of these qualifiers makes it possible to specify addressing modes explicitly.
near ... The accessible addresses range from 00000000H to 00007FFFH and from 0FFFF8000H to 0FFFFFFFFH.
far ... The accessible addresses range from 00000000H to 007FFFFFH and from 0FF800000H to 0FFFFFFFFH.

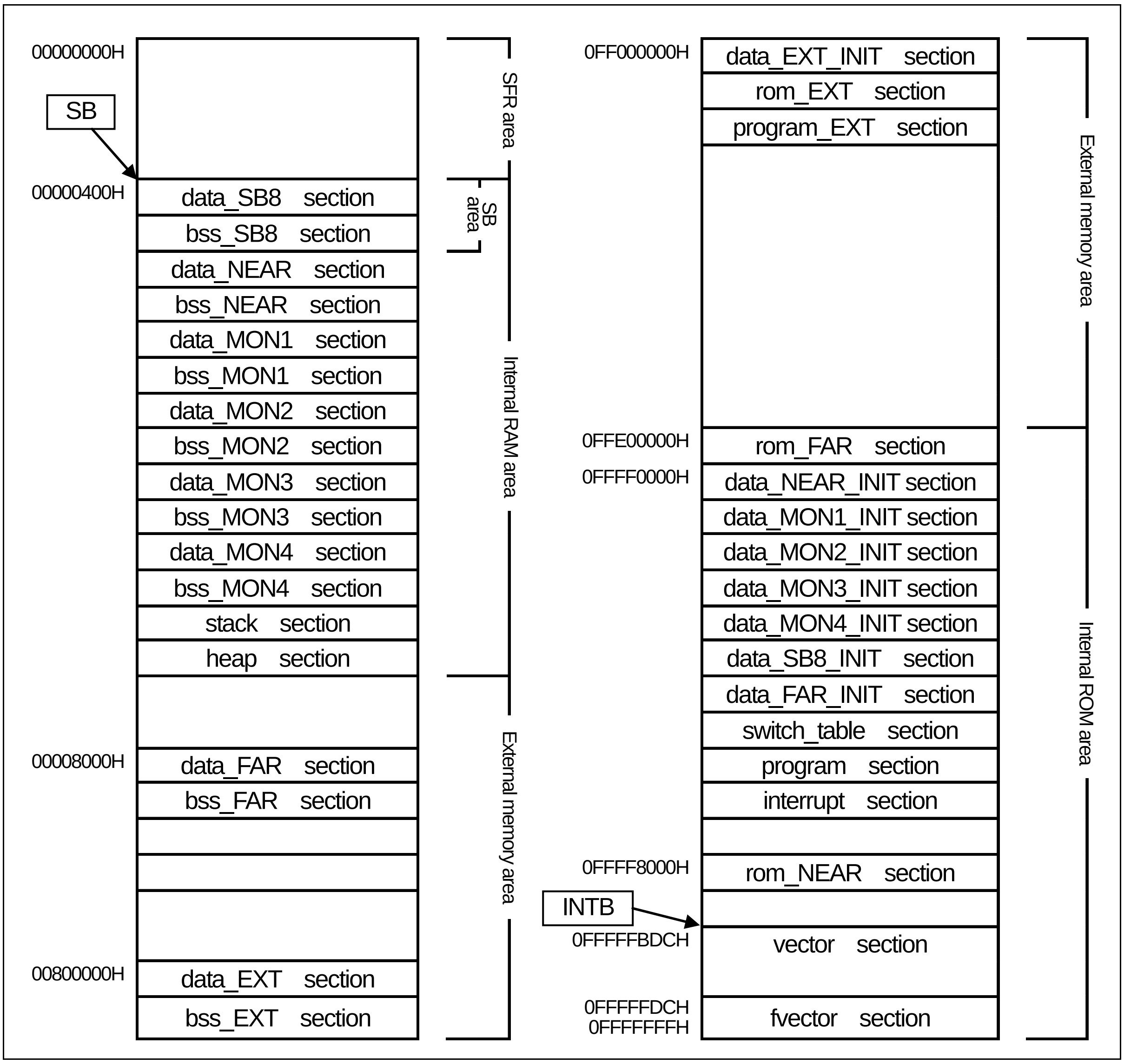


Figure 2.25 Example Section Mapping (1/2)

Also, Figure 2.26 shows the how the sections are mapped according to the sample startup program's include file sect100.inc (used #pragma SB16DATA Extended Functions).
See the "B.7 #pragma Extended Functions" and "2.2.1.f #pragma SB16DATA" for the "#pragma SB16DATA Extended Functions".

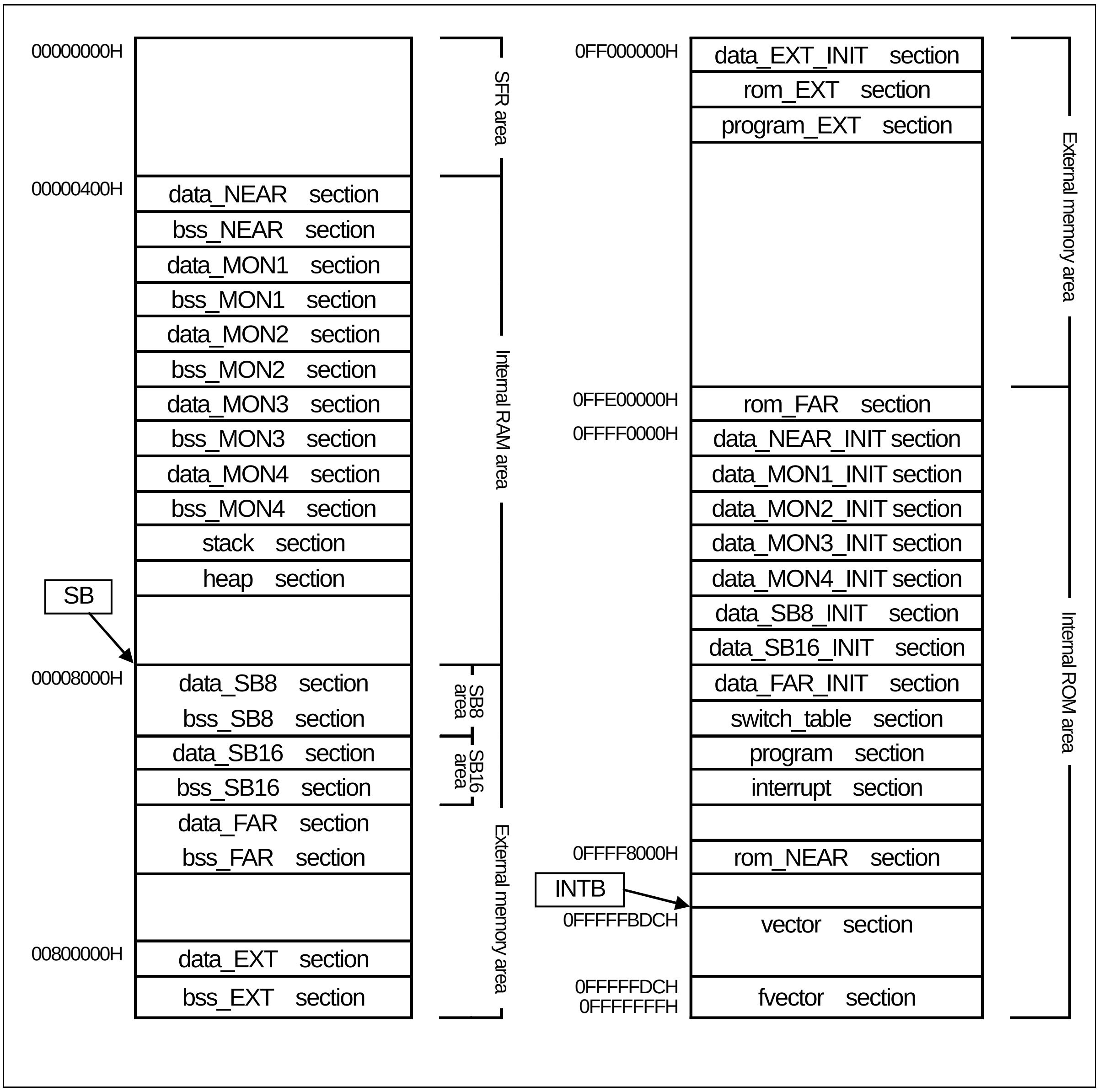


Figure 2.26 Example Section Mapping (2/2)

b Outline of memory mapping setup file

(1) About sect100.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 sect100.inc

Figure 2.27 summarizes the steps required to modify the startup programs to match the target system.

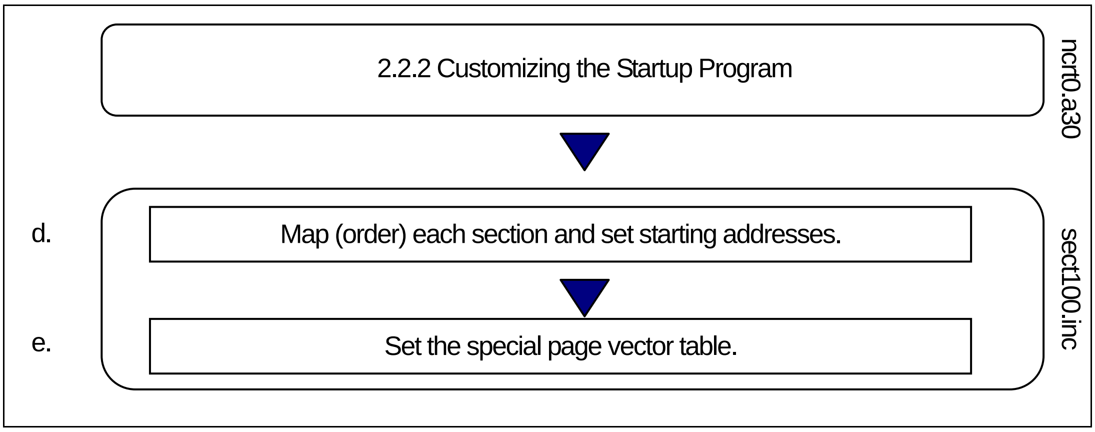


Figure 2.27 Example Sequence for Modifying Startup Programs

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 sect100.inc include file of the startup program.
The sections are mapped to memory in the order they are defined in sect100.inc. Use the as100 pseudo instruction .ORG to specify their starting addresses.
Figure 2.28 is an example of these settings.

```
.section        program,    code, align
.org            0FFFF0000H          ← Specifies the starting address of the program section
```

Figure 2.28 Example Setting of Section Starting Address

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

(1) Rules for Mapping Sections to Memory

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

(1) Sections mapped to RAM

- stack section
- heap section
- data_NEAR section
- bss_NEAR section

- data_FAR section
- data_EXT section
- data_SB8 section
- data_SB16 section
- data_MON1 section
- data_MON2 section
- data_MON3 section
- data_MON4 section
- bss_FAR section
- bss_EXT section
- bss_SB8 section
- bss_SB16 section
- bss_MON1 section
- bss_MON2 section
- bss_MON3 section
- bss_MON4 section

(2) Sections mapped to ROM

- program section
- interrupt section
- switch_table section
- rom_EXT section
- data_NEAR_INIT section
- data_FAR_INIT section
- data_EXT_INIT section
- data_SB8_INIT section
- data_SB16_INIT section
- program_EXT section
- fvector section
- rom_NEAR section
- rom_FAR section
- data_MON1_INIT section
- data_MON2_INIT section
- data_MON3_INIT section
- data_MON4_INIT section

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

(1) Sections mapped only to 0H - 07FFFH, 0FFFF8000H - 0FFFFFFFFH (near area)

- data_NEAR section
- rom_NEAR section
- bss_NEAR section

(2) Sections mapped only to 0H - 07FFFFFH, 0FF800000H - 0FFFFFFFFH (farr area)

- program section
- switch_table section
- data_FAR section
- data_MON1 section
- data_MON2 section
- data_MON3 section
- data_MON4 section
- rom_FAR section
- interrupt section
- bss_FAR section
- bss_MON1 section
- bss_MON2 section
- bss_MON3 section
- bss_MON4 section
- vector section

(3) Sections mapped only to 0FFFFFFDCH - 0FFFFFFFFH

- fvector section

(4) Sections mapped to any area for the R32C/100 series

- stack section
- data_EXT section
- rom_EXT section
- data_NEAR_INIT section
- data_EXT_INIT section
- data_SB16_INIT section
- data_MON2_INIT section
- data_MON4_INIT section
- heap section
- bss_EXT section
- program_EXT section
- data_FAR_INIT section
- data_SB8_INIT section
- data_MON1_INIT section
- data_MON3_INIT section

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

- program_EXT section
- data_NEAR section
- data_FAR section
- data_EXT section
- data_MON1 section
- data_MON2 section
- switch_table section
- data_NEAR_INIT section
- data_FAR_INIT section
- data_EXT_INIT section
- data_MON1_INIT section
- data_MON2_INIT section

- data_MON3 section
- data_MON4 section
- data_SB8 section
- data_SB16 section
- bss_NEAR section
- bss_FAR section
- bss_EXT section
- bss_SB8 section
- bss_SB16 section
- rom_FAR section
- data_MON3_INIT section
- data_MON4_INIT section
- data_SB8_INIT section
- data_SB16_INIT section
- bss_MON1 section
- bss_MON2 section
- bss_MON3 section
- bss_MON4 section
- rom_NEAR section
- rom_EXT section

(2) Example Section Mapping in Single-Chip Mode

Figure 2.29, to Figure 2.32 are examples of the sect100.inc include file which is used for mapping sections to memory in single-chip mode.

```
;*************************************************************************;
;
;    C COMPILER for R32C/100
;    Copyright(C) XXXX. Renesas Technology Corp.
;    and Renesas Solutions Corp., All rights reserved.
                 :
               (omitted)
                 :
;    $Id: sect100.inc,v X.X XXXX/XX/XX XX:XX:XX XXXXX Exp $
;
;*************************************************************************;
;-----------------------------------------------------------------------;
;
;              Arrangement of section
;
;-----------------------------------------------------------------------;
;-----------------------------------------------------------------------;
; NEAR RAM SECTIONS                                                     ;
;-----------------------------------------------------------------------;
                    .section    data_SB8,  data
                    .org                   00000400H
data_SB8_top:
                    .section    bss_SB8,              data, align
bss_SB8_top:
                    .section    data_NEAR,            data, align
data_NEAR_top:
                    .section    bss_NEAR,             data, align
bss_NEAR_top:
                    .section    data_MON1,            data, align
data_MON1_top:
                    .section    bss_MON1,             data, align
bss_MON1_top:
                    .section    data_MON2,            data, align
data_MON2_top:
                    .section    bss_MON2,             data, align
bss_MON2_top:
                    .section    data_MON3,            data, align
data_MON3_top:
                    .section    bss_MON3,             data, align
bss_MON3_top:
                    .section    data_MON4,            data, align
data_MON4_top:
                    .section    bss_MON4,             data, align
bss_MON4_top:

;-----------------------------------------------------------------------;
; STACK SECTION                                                         ;
;-----------------------------------------------------------------------;
                    .section    stack,                data, align
                    .blkb                  STACKSIZE
                    .align
stack_top:
                    .blkb                  ISTACKSIZE
                    .align
istack_top:
```

Figure 2.29   Listing of sect100.inc in Single-Chip Mode (1/4)

```
;---------------------------------------------------------------;
; HEAP SECTION                                                  ;
;---------------------------------------------------------------;
                .section    heap,           data, align
heap_top:
                .blkb               HEAPSIZE

;---------------------------------------------------------------;
; SB RELATIVE RAM SECTIONS                                      ;
;---------------------------------------------------------------;
;               .section    data_SB8,   data
;               .org                    00008000H
;data_SB8_top:
;               .section    bss_SB8,    data, align
;bss_SB8_top:
;               .section    data_SB16,data, align
;data_SB16_top:
;               .section    bss_SB16,   data, align
;bss_SB16_top:

;---------------------------------------------------------------;
; FAR RAM SECTIONS                                              ;
;---------------------------------------------------------------;
                .section    data_FAR,   data, align
data_FAR_top:
                .section    bss_FAR,    data, align
bss_FAR_top:

;---------------------------------------------------------------;
; EXTENDED RAM SECTIONS                                         ;
;---------------------------------------------------------------;
                .section    data_EXT,   data
                .org                    00800000H
data_EXT_top:
                .section    bss_EXT,    data, align
bss_EXT_top:

;---------------------------------------------------------------;
; EXTENDED ROM SECTIONS                                         ;
;---------------------------------------------------------------;
                .section    data_EXT_INIT,      romdata
                .org                    0FF000000H
data_EXT_INIT_top:
                .section    rom_EXT,    romdata, align
rom_EXT_top:
                .section    program_EXT,        code, align

;---------------------------------------------------------------;
; FAR ROM SECTIONS                                              ;
;---------------------------------------------------------------;
                .section    rom_FAR,    romdata
                .org                    0FFE00000H
rom_FAR_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.

Figure 2.30 Listing of sect100.inc in Single-Chip Mode (2/4)

```
;---------------------------------------------------------------;
; INITIAL DATA SECTIONS                                         ;
;---------------------------------------------------------------;
                    .section    data_NEAR_INIT,     romdata
                    .org        0FFFF0000H
data_NEAR_INIT_top:
                    .section    data_MON1_INIT,     romdata, align
data_MON1_INIT_top:
                    .section    data_MON2_INIT,     romdata, align
data_MON2_INIT_top:
                    .section    data_MON3_INIT,     romdata, align
data_MON3_INIT_top:
                    .section    data_MON4_INIT,     romdata, align
data_MON4_INIT_top:
                    .section    data_SB8_INIT,      romdata, align
data_SB8_INIT_top:
;                   .section    data_SB16_INIT,     romdata, align
;data_SB16_INIT_top:
                    .section    data_FAR_INIT,      romdata, align
data_FAR_INIT_top:

;---------------------------------------------------------------;
; SWITCH TABLE SECTIONS                                         ;
;---------------------------------------------------------------;
                    .section    switch_table,       romdata, align

;---------------------------------------------------------------;
; CODE SECTIONS                                                 ;
;---------------------------------------------------------------;
                    .section    program,    code, align

                    .section    interrupt,  code, align

;---------------------------------------------------------------;
; NEAR ROM SECTIONS                                             ;
;---------------------------------------------------------------;
;                   .section    rom_NEAR,           romdata
;                   .org        0FFFF8000H
;rom_NEAR_top:

;---------------------------------------------------------------;
; VARIABLE VECTOR SECTION                                       ;
;---------------------------------------------------------------;
                    .section    vector,             romdata
                    .org        VECTOR_ADR
.if         __MVT__ == 1
                :
            (omitted)
                :
.endif
```

Figure 2.31 Listing of sect100.inc in Single-Chip Mode (3/4)

```
;---------------------------------------------------------------------------;
; FIXED VECTOR SECTION                                                      ;
;---------------------------------------------------------------------------;
                  .section     fvector,     romdata
                  .org                      0FFFFFFDCH
UDI:              .lword                    dummy_int
OVER_FLOW:        .lword                    dummy_int
BRKI:             .lword                    dummy_int
                  .lword                    0FFFFFFFFH
                  .lword                    0FFFFFFFFH
WDT:              .lword                    dummy_int
                  .lword                    dummy_int
NMI:              .lword                    dummy_int
RESET:            .lword                    start

;---------------------------------------------------------------------------;
; ID code    DEFINITION ;
;---------------------------------------------------------------------------;
; ID code check function
; .id "CodeChk"

;****************************************************************************;
;                                                                            ;
;         End of R32C/100 start up                                           ;
;                                                                            ;
;****************************************************************************;
```

Figure 2.32 Listing of sect100.inc in Single-Chip Mode (4/4)

e Setting Interrupt Vector Table

In a program that uses interrupt processing, set up an interrupt vector table by

(1) Setting up the interrupt vector table of the vector section in sect100.inc

The contents of interrupt vectors differ with each microcomputer type. Make sure the interrupt vectors you've set suit the microcomputer type you use. For details, refer to the user's manual of your microcomputer.

(1) When setting up the interrupt vector table in sect100.inc

For programs that use interrupt processing, change the interrupt vector table for the vector section in sect100.inc.

Figure 2.33 shows an example interrupt vector table.

```
;---------------------------------------------------------------;
; VARIABLE VECTOR SECTION                                       ;
;---------------------------------------------------------------;
                .section    vector,         romdata
                .org        VECTOR_ADR
.if     __MVT__ == 1
        .lword  dummy_int       ; BRK                   (software int 0)
           :
        (omitted)
           :
        .lword  dummy_int       ; DMA0                  (software int 8)
        .lword  dummy_int       ; DMA1                  (software int 9)
        .lword  dummy_in        ; DMA2                  (software int 10)
           :
        (omitted)
           :
        .lword  dummy_int       ; uart1 trance/NACK     (software int 19)
        .lword  dummy_int       ; uart1 receive/ACK     (software int 20)
        .lword  dummy_int       ; TIMER B0              (software int 21)
           :
        (omitted)
           :
        .lword  dummy_int       ; INT5                  (software int 26)
        .lword  dummy_int       ; INT4                  (software int 27)
           :
        (omitted)
           :
        .lword  dummy_int       ; uart2 trance/NACK     (software int 33)
        .lword  dummy_int       ; uart2 receive/ACK     (software int 34)
           :
        (omitted)
           :
        .lword  dummy_int       ; software int 255

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

Figure 2.33 Interrupt Vector Address Table

Follow the procedure described below to alter the interrupt vector table of the vector section in sect100.inc.

(1) Externally declare the interrupt processing function in the .GLB as100 pseudo instruction.

(2) The labels of functions created by NC100 are preceded by the underscore (_). Therefore, the names of interrupt processing functions declared here should also be preceded by the underscore.

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

Figure 2.34 is an example of registering the UART1 send interrupt processing function uarttrn.

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

(remainder omitted)
```

Figure 2.34 Example Setting of Interrupt Vector Addresses

Chapter 3 Programming Technique

This chapter describes precautions to be observed when programming with the C compiler, NC100.

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 NC100 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 R32C's Type Dependent Part

Notes about the R32C'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.

```
#pragma ADDRESS   TA0IC   006Ch          /* R32C/100 MCU's Timer A0 interrupt control register */

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

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

Figure3.1 C language description to SFR area

3.1.3 About Optimization

a Regular optimization

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

(1) Meaningless variable access

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

```
extern int  port;

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

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

Although the intended operation in this example is only to read out port, the readout code actually is not optimized before being output. To suppress optimization, add the volatile qualifier as shown in Figure3.2.

```
extern int volatile   port;

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

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

(2) Meaningless comparison

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

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

Figure3.4 eaningless Comparison

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

(3) Programs not executed

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

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

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

Figure3.5 Program Not Executed

(4) Operation between constants

Operation between constants is performed when compiling.

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

        return i;
}
```

Figure3.6 Program Not Executed

(5) Selection of optimum instructions

Selection of optimum instructions as when 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)? */
```

Figure3.7 Example of Ambiguously Interpreted volatile qualifier

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 compile option "-Ono_bit(shortcut -ONB)".

3.1.4 Precautions on Using register Variables

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

If the compile option "-fenable_register(-fER)" is specified, the variables that are register-qualified so as to satisfy specific conditions can be forcibly assigned to registers. This facility is provided for improving generated codes without relying on optimization.
Because improper use of this facility produces negative effects, always be sure to examine generated codes before deciding to use it.

b About register qualification and optimization options

The compiler automatically assigns variables to the registers. This assignment facility is unaffected by a register qualification.

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 Using Prototype declaration Efficiently

NC100 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 NC100, 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 type short type	Expanded into the int type when stacked.
float type	Expanded into the double type when stacked.
otherwise type	Not expanded when stacked.

For this reason, NC100 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.

b Using SB Register Efficiently

Using the SB register-based addressing mode, you can reduce the size of your application program (ROM size). NC100 allows you to declare variables that use the SB register- based addressing mode by writing the description shown in Figure3.8.

```
#pragma SBDATA   val

int          val;
```

Figure3.8 Example of variable declaration using SB-based addressing mode

c Compressing ROM Size Using Compile Option -fJSRW

When calling a function defined outside the file in NC100, the function is called with the JSR.A instruction.
However, if the program is not too large, most functions can be called with the "JSR.W" instruction.
In this case, ROM size will be reduced by doing as follows :
First, Compile with the -fJSRW option and check functions which are indicated as errors at link-time. Then change declarations for the error functions only into declarations using "#pragma JSRA function-name".
When you use the -OGJ option, the JMP instruction at the time of a link is chosen.

d Other methods

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

(1) Change 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 a function which returns a value in only the range of char type, declare its return value type with char.
(4) 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 Figure3.9 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;
```

Figure3.9 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.

```
                ;-------------------------------------------------------;
                ; zero clear BSS                                        ;
                ;-------------------------------------------------------;
;               BZERO           bss_SB8_top,    bss_SB8
;               BZERO           bss_SB16_top, bss_SB16
;               BZERO           bss_NEAR_top, bss_NEAR
;               BZERO           bss_FAR_top,    bss_FAR
;               BZERO           bss_MON1_top, bss_MON1
;               BZERO           bss_MON2_top, bss_MON2
;               BZERO           bss_MON3_top, bss_MON3
;               BZERO           bss_MON4_top, bss_MON4
```

Figure3.10 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.
Figure3.11 is an example of calling assembler function asm_func.

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

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

Figure3.11 Example of Calling Assembler Function Without Parameters(sample1.c)

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

Figure3.12 Compiled result of sample1.c(sample1.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, R6R4, R7R5), 16-bit general-purpose registers (R0, R1, R2, R3, R4, R5, R6, R7), or 8-bit general-purpose registers (R0L, R0H, R1L, R1H, R2L, R2H, R3L, R3H) and address registers (A0, A1, A2, A3).
For 64-bit quantities, the compiler uses the 32-bit general-purpose register pair (R3R1R2R0 and R7R5R6R4) or address register pair (A1A0 and A3A2).

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

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

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

Figure3.13 is an example of using #pragma PARAMETER when calling the assembler function asm_func.

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

              asm_func(i, j);
}
```

Figure3.13 Example of Calling Assembler Function With Parameters (sample2.c)

```
              .SECTION  program,CODE,ALIGN
              ._file        'smp2.c'
              .align
              ._line        5
;## # C_SRC :               {
              .glb          _main
_main:
              ._line        6
;## # C_SRC :                             short i = 0x02;
              mov.w         #0002H,R0 ;   i                  ← Parameters are passed via the R0 and R1
              ._line        7                                   registers to the assembler function.
;## # C_SRC :                             short j = 0x05;
              mov.w         #0005H,R1 ;   j
              ._line        9
;## # C_SRC :                             asm_func(i, j);
              jsr           _asm_func                        ← Calls assembler function(preceded by '_')
              ._line        10
;## # C_SRC :               }
              rts
```

Figure3.14 Compiled result of sample2.c(sample2.a30)

c Limits on Parameters in #pragma PARAMETER Declaration

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

- structure types and union type parameters

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

3.3.2 Writing Assembler Functions

a Method for writing the called assembler functions

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

(1) Specify section names using the assembler pseudo-command .SECTION.
(2) Global specify function name labels using the assembler pseudo-command .GLB.
(3) Add the underscore (_) to the function name to write it as label.
(4) When modifying the B and U flags within the function, save the flag register to the stack beforehand.[1]

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

(1) If you modified the B and U flags within the function, restore the flag register from the stack.
(2) 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.

Figure3.15 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 NC100.

```
          .section  program, code, align      ← (1)
          .glb      _asm_func                 ← (2)
_asm_func:                                    ← (3)
          pushc     FLG                       ← (4)
          mov.l     SYM1, R3R1

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

Figure3.15 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. Figure3.16 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
short type	R0 register
int type	R2R0 register (default) R0 register ("-fint_16" option use)
long type	R2R0 register
long long type	The 32 low-order bits are stored in the R2R0 register and the 32 high-order bits are stored in the R3R1 register as the value is returned.
float type	R2R0 register
double type long double type	The 32 low-order bits are stored in the R2R0 register and the 32 high-order bits are stored in the R3R1 register as the value is returned.
pointer type	A0 register
struct type union type	Immediately before calling the function, the far address indicating the area for storing the return value is pushed to the stack. Before the return to the calling program, the called function writes the return value to the area indicated by the far address pushed to the stack.

```
            .section    program
            .glb        _asm_func
_asm_func:
               :
            (omitted)
               :
            mov.l       #01A000H, R2R0
            rts
            .END
```

Figure3.16 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.

Figure3.17 is an example of referencing the C program's global variable counter from the assembler function asm_func.

```
C program:

unsigned short          counter;            ← C program global variable

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

Assembler function:

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

Figure3.17 Referencing a C Global Variable

d Notes on Coding Interrupt Handing in Assembler Function

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

(1) Save the registers (R2R0, R3R1, R6R4, R7R5, A0, A1, A2 and A3) at the entry point.
(2) Restore the registers (R2R0, R3R1, R6R4, R7R5, A0, A1, A2, and A3) at the exit point.
(3) Use the REIT instruction to return from the function.

Figure3.18 is an example of coding an assembler function for interrupt processing.

```
            .section    program
            .glb        _func
_int_func:
            pushm       R2R0,R3R1,R6R4,R7R5,A0,A1,A2,A3     ← Save registers
            mov.b       #01H, R0L
                :
            (omitted)
                :
            popm        R2R0,R3R1,R6R4,R7R5,A0,A1,A2,A3     ← Restore registers
            reit                                            ← Return to C program
            .END
```

Figure3.18 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.

c Notes on Handling General-purpose and Address Registers

The general-purpose registers (R2R0, R3R1, R6R4, and R7R5) and address registers (A0, A1, A2, and A3) can have their contents modified in assembler functions without a problem.

d Passing Parameters to an Assembler Function

Use the #pragma PARAMETER function if you need to pass parameters to a function written in assembly language. The parameters are passed via registers.
Figure3.19 shows the format (asm_func in the figure is the name of an assembler function).

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

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

Figure3.19 Prototype declaration of assembler function

#pragma PARAMETER passes arguments to assembler functions via 32-bit general-purpose registers (R2R0, R3R1, R6R4, and R7R5), 16-bit general-purpose registers (R0, R1, R2, R3, R4, R5, R6, and R7), 8-bit general-purpose registers (R0L, R0H, R1L, R1H, R2L, R2H, R3L, and R3H), and address registers (A0, A1, A2, and A3). In addition, the 32-bit general-purpose registers are combined to form 64-bit registers (R3R1R2R0, R7R5R6R4, A1A0, and A3A2) 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 struct and union types in a #pragma PARAMETER declaration.
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

NC100 basically is compatible with Renesas C compilers "NCxx" at the language specification level (including extended functions). However, there are some differences between the compiler (this manual) and other NC-series compilers as described below.

a Difference in default near/far

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

Table 3.3 Default near/far in the NC Series

Compiler	RAM data	ROM data	Program
NC100	near (However, pointer type is far Fixed)	far	far Fixed
NC308	near (However, pointer type is far)	far	far Fixed
NC30	near	far	far Fixed

Appendix A Command Option Reference

This appendix describes how to start the compile driver nc100 and the command line options. The description of the command line options includes those for the as100 assembler and ln100 linkage editor, which can be started from nc100.

A.1 nc100 Command Format

```
% nc100△[command-line-option]△<[assembly-language-source-file-name]△
                    [relocatable-object-file-name]△[C-source-file-name]>

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

Figure A.1 nc100 Command Line Format

```
% nc100  -osample  -as100  "-l"  -ln100  "-ms"  ncrt0.a30  sample.c<RET>

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

Figure A.2 Example nc100 Command Line

A.2 nc100 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.
-dsource (Short form -dS)	Generates an assembly language source file (extension ".a30") with a C language source list output as a comment. (Not deleted even after assembling.)
-dsource_in_list (Short form -dSL)	In addition to the "-dsource(-dS)" function, generates an assembly language list file (.lst).
-E	Invokes only preprocess commands and outputs result to standard output.
-I*directory*	Specifies the directory containing the file(s) specified in #include. You can specify up to 50 directories.
-P	Invokes only preprocess commands and creates a file (extension .i).
-S	Creates an assembly language source file (extension .a30) and ends processing.
-silent	Suppresses the copyright message display at startup.
-Upredefined macro	Undefines the specified predefined macro.

-c

Compile driver control

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

Notes: If this option is specified, no absolute module file (extension .x30) or other file output by ln100 is created.

-D*identifier*

Compile driver control

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

Syntax: nc100△-D*identifier*[=*constant*]△<C source file>

[= *constant*] is optional.

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

[1] If you do not specify command line options -c, -E, -P, or -S, nc100 finishes at ln100 and output files up to the absolute load module file (extension .x30) are created.

-dsource -dS

Comment option

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

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

-dsource_in_list -dSL

List File option

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

-E

Compile driver control

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

Notes: When this option is specified, no assembly source file (extensions .a30), re-locatable object files (extension .r30), absolute module files (extension .x30), or other files output by ccom100, as100, or ln100 are generated.

-I*directory*

Compile driver control

Function: Specifies the directory name in which to search for files to be referenced by the preprocess command #include.
Max specified 50 directory.

Syntax: nc100△-I*directory*△<C source file>

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

-P

Compile driver control

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

Notes:
(1) When this option is specified, no assembly source file (extensions .a30), re-locatable object files (extension .r30), absolute module files (extension .x30) or other files output by ccom100, as100, or ln100 are generated.
(2) The file (extension .i) generated by this option does not include the #line command generated by the preprocessor. To get a result that includes #line, try again with the -E option.

-S

Compile driver control

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

Notes: When this option is specified, no relocatable object files (extension.r30), absolute module files (extension .x30) or other files output by as100 or ln100 are generated.

-silent

Compile driver control

Function: Suppresses the display of copyright notices at startup.

-U*predefined macro*

Compile driver control

Function: Undefines predefined macro constants.

Syntax: nc100△-U*predefined macro*△<C source file>

Notes: The maximum number of macros that can be undefined may be limited by the maximum number of characters that can be specified on the command line of the operating system of the host machine.
STDC, _LINE_, _FILE_, _DATE_, and _TIME_ cannot be undefined.

A.2.2 Options Specifying Output Files

Table A.2 shows the command line option that specifies the name of the output absolute module file.

Table A.2 Options for Specifying Output Files

Option	Function
-dirdirectory-name	Specifies the destination directory of the file(s) (absolute module file, map file, etc.) generated by ln100.
-o*file-name*	Specifies the name(s) of the file(s) (absolute module file, map file, etc.) generated by ln100. This option can also be used to specify the destination directory. This option can also be used to specify the file name includes the path. Do not specify the filename extension.

-dir*directory-name*

Output file specification

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

Syntax: nc100△-dir*directory-name*

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

-o*file-name*

Output file specification

Function: Specifies the name(s) of the file(s) (absolute module file, map file, etc.) generated by ln100. This option can also be used to specify the file name includes the path.
You must not specify the filename extension.

Syntax: nc100△-o*file-name*△<C source file>

A.2.3 Version Information Display Option

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

Table A.3 Options for Displaying Version Data

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

-v

Display command program name

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

Notes: Use lowercase v for this option.

-V

Display version data

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

Supplement: Use this option to check that the compiler has been installed correctly. The "R32C/100 Series C Compiler package Release Notes" list the correct version numbers of the commands executed internally by the compiler.

If the version numbers in the Release Notes do not match those displayed using this option, the package may not have been installed correctly. See the "R32C/100 Series C Compiler package Release Notes" for details of how to install the NC100 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 lists the debugging startup options that output C language level debug information.

Table A.4 Options for Debugging

Option	Option
-g	Outputs debug information to an assembly language source file (extension .a30). This makes C level debugging of programs possible.
-genter	Always outputs an enter instruction when calling a function. Be sure to specify this option when using the debugger's stack trace function.

-g

Outputting debugging information

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

Notes: When debugging your program at the C language level, always specify this option. Specification of this option does not affect the code generated by the compiler.

-genter

Outputting enter instruction

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

Notes:

(1) When using the debugger's stack trace function, always specify this option. Without this option, you cannot obtain the correct result.

(2) When this option is specified, the compiler generates code to reconstruct the stack frame using the enter command at entry of the function regardless of whether or not it is necessary. Consequently, the ROM size and the amount of stack used may increase.

A.2.5 Optimization Options

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

Table A.5 Optimization Options

Option	Short form	Function
-O[1-5]	None	Optimization of speed and ROM size.
-O5OA	None	Inhibits code generation based on bit-manipulating instructions when the optimization option "-O5" is selected.
-OR	None	Optimization of ROM size followed by speed.
-OS	None	Optimization of speed followed by ROM size.
-OR_MAX	-ORM	Maximum optimization of ROM size followed by speed.
-OS_MAX	-OSM	Maximum optimization of speed followed by ROM size.
-Ocompare_byte_to_word	-OCBTW	Compares consecutive bytes of data at contiguous addresses in words.
-Oconst	-OC	Performs optimization by replacing references to the const-qualified external variables with constants.
-Ofile_inline	-OFI	All inline functions are expanded inline.
-Oinline_line	-OIL	This option changes the size (number of lines) of the function to be inline expanded.
-Oglb_jmp	-OGJ	Global jump is optimized.
-Oglobal_to_inline	-OGTI	Handles global functions as inline-declared.
-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_bit	-ONB	Suppresses optimization based on grouping of bit manipulations.
-Ono_break_source_debug	-ONBSD	Suppresses optimization based on grouping of bit manipulations.
-Ono_float_const_fold	-ONFCF	Suppresses the constant folding processing of floating point numbers.
-Ono_logical_or_combine	-ONLOC	Suppresses the optimization that puts consecutive OR together.
-Ono_asmopt	-ONA	Inhibits starting the assembler optimizer "aopt100".
-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.
-Ostatic_to_inline	-OSTI	A static function is treated as an inline function.

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

Table A.6 Effect of each Optimization Options

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

-O1: Makes "-O3", "-Ono_bit", "-Ono_break_source_debug" and, "-Ono_float_const_fold" valid.

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

-O3: Optimizes speed and ROM size to the maximum.

-O4: "-O3" and "-Oconst" valid.

-O5: Effect the best possible optimization in common sub expressions (if the option "-OR" is concurrently specified); 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.
- When these variables are used in one and the same function.

Exsample :

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

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

result:

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

-O[1-5]

Optimization

Notes:

When the "-O5" optimizing options is used, the compiler generates in some cases "BTSTC" or "BTSTS" bit manipulation instructions. In R32C/100, 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.

```
#pragma ADDRESS ta0ic_addr      006CH /* 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".
  When you use the optimization option of "-O5", please use together with "-O5A."
- An instruction is directly described in a program using an ASM function.

-O5OA

Optimization

Function:

Inhibits code generation based on bit-manipulating instructions when the optimization option "-O5" is selected.

-OR

Optimization

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

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

-OS

Optimization

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

-OR_MAX -ORM

Optimization

Function: Optimizes ROM size in preference to speed.
When this option is used, the effect is same with "-O5", "-O5OA", "-OGJ", "-OR", "-fD32", "-fNA", "-fUF" options.

-OS_MAX -OSM

Optimization

Function: Although the ROM size may somewhat increase, optimization is performed to obtain the fastest speed possible.
When this option is used, the effect is same with "-O4", "-OGJ", "-OGTI", "-OS", "-OSA", "-OSTI", "-OLU=10", "-fD32", "-fUF" options.

-Ocompare_byte_to_word -OCBTW

Optimization

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

-Oconst　　　-OC

Optimization

Function:

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

Supplement:

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

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

-Ofile_inline [= inline expansion file(,...)]　　　-OFI [= inline expansion file(,...)]

Inline expansion

Function:

- All inline functions are expanded in-line.
- Code generation for unreferenced static functions is suppressed.
- If an inline expansion file is specified, inline expansion is performed on global functions extending across a file boundary.
- The inline expansion of global functions extending across a file boundary is performed on only those functions whose expanded size (from '{' to '}') is within 150 lines including a comment line. Note that the size can be changed using the option "-Oinline_line."

Supplement:

Although it normally is necessary that an inline function be declared before its entity can be defined, use of this option permits the entity of an inline function to be defined before the inline function is declared.

- The following shows an example of a program fragment where a function is inline expanded for the option "-Ofile_inline" specified in it.

```
extern int        i;
inline int func(void)

void   main(void)
{
       int        s;

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

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

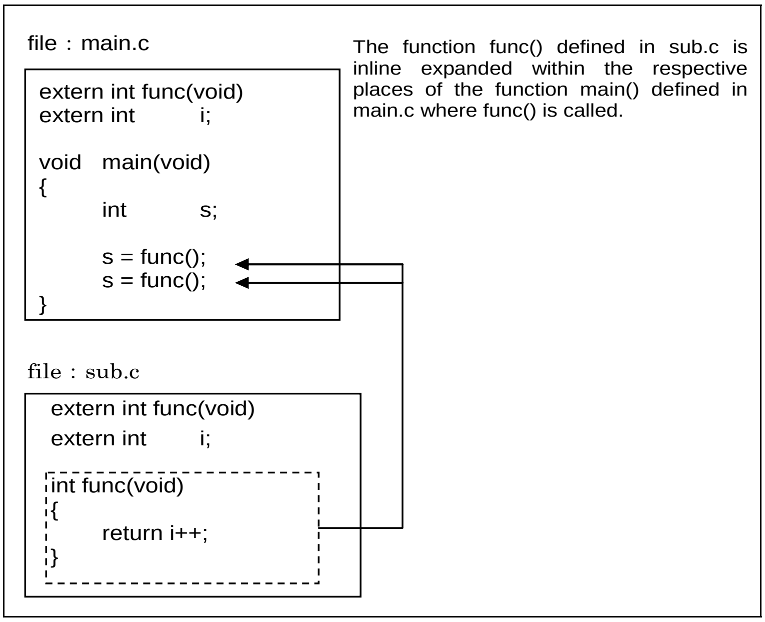
The function func() is inline expanded within the respective places of the function main() in which it is called.

Furthermore, if an inline expansion file is specified, the global functions extending across a file boundary can be inline expanded.

- An example of a program fragment where a function is inline expanded for the option "-Ofile_inline = inline expansion file" specified in it. In the example below, inline expansion is performed on the source file main.c for the option "-Ofile_inline = sub.c" specified in it.

-Ofile_inline [=inline expansion file(,...)] -OFI [=inline expansion file(,...)]

Inline expansion



Notes:

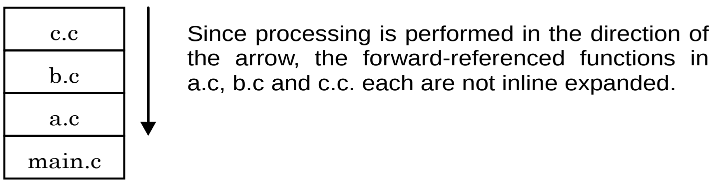
(1) Declaration of an inline function and the definition of the entity of the inline function must be written in one and the same file.

(2) No structures or unions can be used for the arguments to an inline function. If this precaution is neglected, a compile error may result.

(3) Inline functions cannot be called indirectly. If any indirect call is encountered, a compile error may result.

(4) Inline functions cannot be called recursively. If any recursive call is encountered, a compile error may result.

(5) If multiple inline expansion files are specified, inline expansion is performed in the order in which the files are specified. If the inline expand option is specified as "main.c -Ofile_inline = a.c, b.c, c.c," it is processed assuming the file configuration shown below.



(6) This option applies to the program section only. If section names are changed by #pragma SECTION, functions are not inline expanded across a file boundary.

(7) The static functions defined by #pragma __ASMMACRO that begin with the underbar (_) (those defined in asmmacro.h and string.h) are inline expanded.

-Oinline_line = inline expansion line -OI L= inline expansion line

Inline expansion

Function:

This option changes the size (number of lines) of the function to be inline expanded for the option "-Ofile_inline," "-Oglobal_to_inline" or "-Ostatic_to_inline" specified in the program.

When this option is omitted, inline expansion is performed on only the functions whose expanded size from '{' to '}' is within 150 lines including a comment line.

-Oglb_jmp -OGJ

Optimization

Function: Global jump is optimized.

-Oglobal_to_inline -OGTI

Handles global functions as inline-declared.

Function: Handles global functions as inline-declared functions (inline functions) to generate inline-expanded assemble code.

Supplement: When the following conditions are met, the compiler handles global functions as inline functions to generate inline-expanded assemble code.

(1) Operation is performed on the global function whose body is written prior to a function call.
- A function call and the body of that function must be written in the same source file.
- If the "-Ofile_inline" option is selected, this condition should be ignored.

(2) If no addresses are acquired in the program for the target global function

(3) If the target global function is recursively called

The following shows an example of how a global function to be inline expanded will be written.

```
extern int      i;

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

void    main(void)
{
        int     s;

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

The function func() is inline expanded in respective places of the function main() in which it is called.

Notes:

(1) Assembler code for the body of the global function handled as an inline function is always generated.

(2) If a function needs to be forcibly handled as an inline function, be sure to declare it as inline.

(3) Inline expansion is performed on only the functions whose expanded size from '{' to '}' is within 150 lines including a comment line. Note that the size can be changed using the option "-Oinline_line."

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

Unrolls a loop

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

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

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

-Ono_bit -ONB

Suppression of optimization

Function: Suppresses optimization based on grouping of bit manipulations.

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:
(1) This optimization is performed, The variables is specified regardless volatile qualified.
(2) 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.

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.

Supplement: By default, NC100 folds constants. Following is an example.

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

after optimization:
        val/20e250
```

In this case, if the application uses the full dynamic range of floating points, the results of calculation differ as the order of calculation is changed. This option suppresses the constant folding in floating point numbers so that the calculation sequence in the C source file is preserved.

-Ono_logical_or_combine -ONLOC

Suppression of optimization

Function: Suppresses the optimization that puts logical OR together.

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

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

                 ↓ (Optimized)

        if( a & 0x07 )
```

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

-Ono_asmopt -ONA

Inhibits starting the assembler optimizer

Function: Inhibits starting the assembler optimizer "aopt100".

-Osp_adjust -OSA

Removing stack correction code after calling a function

Function: Optimizes code generation by combining stack correction codes after function calls. Please use this option together with -O[1-5].

Supplement: Because the area for arguments to a function normally is deal located 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.

```
char     func1(char, char, char);
char     func2(char, char, char);

void     main( void ) {
         char     i = 1;
         char     j = 2;
         char     k=3;
         char     l, m;

         l   = func1( i, j, k);
         m = func2( i, j, 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.

-Ostatic_to_inline -OSTI

A static function is treated as an inline function

Function:

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

Supplement:

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

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

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

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

The following shows an example of how a static function to be inline expanded will be written.

```
extern int        i;

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

void    main(void)
{
        int       s;

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

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

Notes:

(1) The assembler code to description of substance of the static function which became inline function treatment is always generated.
However, it is not generated when using it together with the option "- Ofile_ inline".

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

(3) Inline expansion is performed on only the functions whose expanded size from '{' to '}' is within 150 lines including a comment line. Note that the size can be changed using the option "-Oinline_line."

A.2.6 Generated Code Modification Options

Table A.7 shows the command line options for controlling nc100-generated assembly code.

Table A.7 Generated Code Modification Options

Option	Short form	Function
-fansi	None	Makes "-fnot_reserve_far_and_near", "-fnot_reserve_asm", and "-fextend_to_int" valid.
-fconst_not_ROM	-fCNR	Does not handle the types specified by const as ROM data.
-fdouble_32	-fD32	This option specifies that the double type be handled in 32-bit data length as is the float type.
-fenable_register	-fER	Make register storage class available.
-fextend_to_int	-fETI	Performs operation after extending char-type data to the int type. (Extended according to ANSI standards.) [1]
-ffar_RAM	-fFRAM	Changes the default attribute of RAM data to far.
-finfo	None	Outputs the information required for the Inspector, Call Walker and Map Viewer to the absolute module file (.x30).
-fint_16	-fI16	Does handle int type at the 16-bit width.
-fJSRW	None	Changes the default instruction for calling functions to JSR.W.
-fnear_ROM	-fNROM	Changes the default attribute of ROM data to near.
-fno_align	-fNA	Does not align the start address of the function.
-fno_switch_table	-fNST	When this option is specified, the code which branches since it compares is generated to a switch statement.
-fnot_address_volatile	-fNAV	Does not regard the variables specified by #pragma ADDRESS (#pragma EQU) as those specified by volatile.
-fnot_reserve_asm	-fNRA	Exclude asm from reserved words. (Only _asm is valid.)
-fnot_reserve_far_and_near	-fNRFAN	Exclude far and near from reserved words. (Only _far and _near are valid.)
-fnot_reserve_inline	-fNRI	Exclude far and near from reserved words. (Only _inline is made a reserved word.)
-fsigned_char	-fSC	Handles type char without sign specification as type signed char.
-fswitch_other_section	-fSOS	This option outputs a ROM table for a 'switch' statement to some other section than a program section.
-fuse_FPU	-fUF	Outputs FPU instruction.

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

-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

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

-fconst_not_ROM -fCNR

Modify generated code

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

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

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

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

-fdouble_32 -fD32

Modify generated code

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

Supplement:
(1) When specifying this option, always make sure the prototype of the function is declared. If no prototype declarations exist, invalid code may be generated.
(2) When this option is selected, the debug information for type double is handled as type float. In the C watch window or global window, etc. of the emulator debugger or simulator debugger, said information is displayed as type float.

-fenable_register — -fER

Register storage class

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

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

- Integral type variable
- Floating point variable
- Pointer variable

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

-fextend_to_int — -fETI

Modify generated code

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

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

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

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

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

-ffar_RAM　-fFRAM

Modify generated code

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

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

-finfo

Modify generated code

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

Supplement: When using "Call Walker" and "Map Viewer" the absolute module file ".x30" output by this option is needed.

-fint_16　-fI16

Modify generated code

Function: Does handle int type at the 16-bit width.

Supplement: When using this option, you need to link nc100i16.lib instead of nc100lib.lib as the standard library. If you executed a range of operations from compile to link after specifying this option from the compiler driver, the libraries to be linked are automatically changed.
The default size of type int when this option is not specified is 32 bits.
Note that if this option is used in combination with the compile option "-fuse_FPU(-fIUF)," you need to link nc100i16fpu.lib instead of nc100lib.lib.

-fJSRW

Modify generated code

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

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

Notes: 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".

-fnear_ROM -fNROM

Modify generated code

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

Supplement: The ROM data (const-specified variables, etc.) are located in the far area by default. By specifying this option you can locate the ROM data in the near area.

-fno_align -fNA

Modify generated code

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

-fno_switch_table -fNST

Modify generated code

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

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

-fnot_address_volatile -fNAV

Modify generated code

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

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

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

-fnot_reserve_asm -fNRA

Modify generated code

Function: Removes asm from the list of reserved words.

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

-fnot_reserve_far_and_near -fNRFAN

Modify generated code

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

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

-fnot_reserve_inline -fNRI

Modify generated code

Function: Does not handle inline as a reserved word.

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

-fsigned_char -fSC

Modify generated code

Function: Handles type char without sign specification as type signed char.

-fswitch_other_section -fSOS

Modify generated code

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

Supplement: Section name is 'switch_table'

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

-fuse_FPU -fUF

Modify generated code

Function: Outputs FPU instruction.

Supplement: When using this option, you need to link nc100fpu.lib instead of nc100lib.lib as the standard library. If you executed a range of operations from compile to link after specifying this option from the compiler driver, the libraries to be linked are automatically changed.
Note that if this option is used in combination with the compile option "-fint_16(-fI16)," you need to link nc100i16fpu.lib instead of nc100lib.lib.

A.2.7 Library Specifying Option

Table A.8 lists the startup options you can use to specify a library file.

Table A.8 Library Specifying Option

Option	Function
-l*libraryfilename*	Specifies a library file that is used by ln100 when linking files.

-llibrary-file-name

Function: Specifies a library file that is used by ln100 when linking files. The file extension can be omitted.

Syntax: nc100△-l*filename*△<*C source file name*>

Notes:

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

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

(3) NC100 links by default the library "nc100lib.lib" that is present in the directory specified by the environment variable LIB100. The table below lists the library files to be linked for each compile option specified.

compile option "-fint_16"	compile option "-fuse_FPU"	reference library
None	None	nc100lib.lib
Specify	None	nc100i16.lib
None	Specify	nc100fpu.lib
Specify	Specify	nc100i16fpu.lib

(4) If multiple libraries are specified, references to "nc100lib.lib" are assigned the lowest priority.

A.2.8 Warning Options

Table A.9 shows the command line options for outputting warning messages for contraventions of nc100 language specifications.

Table A.9 Warning Options

Option	Short form	Function
-Wall	None	Displays message for all detectable warnings. (however, not including alarms output by -Wlarge_to_small and "-Wno_used_argument")
-Wccom_max_warnings =*Warning Count*	-WCMW	This option allows you to specify an upper limit for the number of warnings output by ccom100.
-Werror_file<*file name*>	-WEF	Outputs error messages to the specified file.
-Wignore_near_pointer	-WINP	Inhibits a warning when the near pointer is handled as a far pointer.
-Wlarge_to_small	-WLTS	Outputs a warning about the tacit transfer of variables in descending sequence of size.
-Wmake_tagfile	-WMT	If an error or warning occurred, a tag file is output for each file.
-Wnesting_comment	-WNC	Outputs a warning for a comment including "*/" .
-Wno_stop	-WNS	Prevents the compiler stopping when an error occurs.
-Wno_used_argument	-WNUA	Outputs a warning for unused argument of functions.
-Wno_used_function	-WNUF	Displays unused global functions when linking.
-Wno_used_static_function	-WNUSF	For one of the following reasons, a static function name is output that does not require code generation.
-Wno_warning_stdlib	-WNWS	Specifying this option while "-Wnon_prototype" or "-Wall" is specified inhibits "Alarm for standard libraries which do not have prototype declaration.
-Wnon_prototype	-WNP	Outputs warning messages for functions without prototype declarations.
-Wstdout	None	Outputs error messages to the host machine's standard output (stdout).
-Wstop_at_link	-WSAL	Stops linking the source files if a warning occurs during linking to suppress generation of absolute module files. Also, a return value "10" is returned to the host OS.
-Wstop_at_warning	-WSAW	Stops compiling the source files if a warning occurs during compiling and returns the compiler end code "10".
-Wundefined_macro	-WUM	Warns you that undefined macros are used in #if.
-Wuninitialize_variable	-WUV	Outputs a warning about auto variables that have not been initialized.
-Wunknown_pragma	-WUP	Outputs warning messages for non-supported #pragma.
-Wmultiple_tentative_definitions	-WMTD	Outputs a warning when there are multiple tentative definitions for one and the same variable name.

-Wall

Warning Options

Function: Indicates all detectable alarms.

Supplement:

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

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

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

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

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

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

Warning Options

Function: This option allows you to specify an upper limit for the number of warnings output by ccom100.

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

Notes For the upper-limit count of warning outputs, specify a number equal to or greater than 0. Specification of this count cannot be omitted. When you specify 0, warning outputs are completely suppressed inhibited.

-Werror_file <*file-name*>

Warning Options

Function: Outputs error messages to the specified file.

Syntax: nc100△ -W*error_file*△<*output error message file name*>

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

-Wignore_near_pointer -WINP

Warning Options

Function: Inhibits a warning when the near pointer is handled as a far pointer.

Supplement: In the compiler, the pointer attribute is fixed to far (32 bits). The compiler by default ignores near qualifiers for the pointer after generating a warning.
If this option is specified, the compiler inhibits a warning that near qualifiers for the pointer are ignored.

-Wlarge_to_small -WLTS

Warning Options

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

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

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

- Assignment from char type variables to char type variables
- Assignment of immediate values to char type variables
- Assignment of immediate values to float type variables

-Wmake_tagfile -WMT

Warning Options

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

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

-Wnesting_comment -WNC

Warning Options

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

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

-Wno_stop -WNS

Warning Options

Function: Prevents the compiler stopping when an error occurs.

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

Notes: A system error may occur due to erroneous description in the program. In such a case, the compiler stops compiling even when this option is specified.

-Wno_used_argument -WNUA

Warning Options

Function: Outputs a warning for unused arguments function.

-Wno_used_function -WNUF

Warning Options

Function: Displays unused global functions when linking.

Notes: When selecting this option, be sure to specify the "-finfo" option at the same time.

-Wno_used_static_function -WNUSF

Warning Options

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

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

Supplement: Code generation for the static functions output by the compiler when this option is specified can be suppressed by specifying the "-ferase_static_function(-fESF)" option.

Notes:

(1) If you want to suppress code generation for the static functions output by the compiler when the "-Ostatic_to_inline(-OSTI)" option is specified, be sure to use "-ferase_static_function(-fESF)" to suppress code generation(The entities of the static functions output by the compiler when this option is specified can never be deleted from the C source file.).

(2) If a function name is written for the initializer of an array as shown below, the compiler handles the function as referenced even though it may not actually be referenced during program operation. In the example given below, although the functions f4 and f5 are not referenced, the compiler handles them as referenced.

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

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

-Wno_warning_stdlib -WNWS

Warning Options

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

-Wnon_prototype -WNP

Warning Options

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

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

-Wstdout

Warning Options

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

Supplement: Use this option to save error output, etc. to a file by using Redirect in the Microsoft Windows (TM).

Notes: In this Compiler for Microsoft Windows (TM), errors from as100 and ln100 invoked by the compile-driver are output to the standard output regardless of this option.

-Wstop_at_link -WSAL

Warning Options

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

-Wstop_at_warning -WSAW

Warning Options

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

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

-Wundefined_macro -WUM

Warning Options

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

-Wuninitialize_variable -WUV

Warning Options

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

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

-Wunknown_pragma -WUP

Warning Options

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

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

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

-Wmultiple_tentative_definitions -WMTD

Warning Options

Function: Outputs a warning when there are multiple tentative definitions for one and the same variable name.

Supplement: If variables are declared outside a function by not using an initializer and without a storage class specifier or with storage class static, such a declaration is referred to as "tentative definition."
If this option is specified, the compiler outputs a warning when such a declaration is encountered two or more times.

A.2.9 Assemble and Link Options

Table A.10 shows the command line options for specifying as100 and ln100 options.

Table A.10 Assemble and Link Options

Option	Function
-as100△< Option>	Specifies options for the as100 link command. If you specify two or more options, enclose them in double quotes.
-ln100△< Option>	Specifies options for the ln100 assemble command. If you specify two or more options, enclose them in double quotes.

-as100 "*Option*"

Assemble/link option

Function: Specifies as100 assemble command options
If you specify two or more options, enclose them in double quotes.

Syntax: nc100△-as100△"*option1*△*option2*"△<*C source file*>

Notes: Do not specify the as100 options "-.", "-C", "-O", -PSFP", "-T", or "-V".

-ln100 "*Option*"

Assemble/link option

Function: Specifies options for the ln100 link command. You can specify a maximum of four options.
If you specify two or more options, enclose them in double quotes.

Syntax: nc100△-ln100△"*option1*△*option2*"△<*C source file name*>

Notes: Do not specify the ln100 options "-.", "-G", "-O", "-ORDER", "-L", "-T", "-V" or "@ file".

A.3 Notes on Command Line Options

A.3.1 Coding Command Line Options

The NC100 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 NC100 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 R32C/100 series, NC100 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	Specifies the addressing mode to access data. near.....Access to an area within 64K bytes (00000000H-00007FFFH and 0FFFF8000H-0FFFFFFFFH). far.......Access to an area within 4G bytes (00000000H-007FFFFFH and 0FF800000H-0FFFFFFFFH). ● 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	(4) Permits you to use Japanese characters in character strings. Example： L"漢字" (5) Permits you to use Japanese characters for character constants. Example： L'漢' (6) 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	Default value can be defined for the argument of a function.
	Example：
	`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.

Table B.2 Extended Functions (2/2)

Extended feature	Description
Inline storage class	Functions can be inline developed by using the inline storage class specifier.inline. Example： inline func(int i); ● Always be sure to define the body of an inline function before using the inline function.
Extension of Comments	You can include C++-like comments ("//"). Example： // This is a comment.
#pragma Extended functions	You can use extended functions for which the hardware of R32C/100 series in C language.

macro assebler function	You can describe some assembler command as the function of C
	Example 1：
	signed char abs_b(signed char val);
	Example 2：
	long int abs_l(long int val);

B.1 Near and far Modifiers

For the R32C/100 series microcomputers, the addressing modes used for referencing and locating data vary around the boundary address 00007FFFH and 0FFFF8000H. NC100 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 00000000H-00007FFFH and 0FFFF8000H-0FFFFFFFFH
- far modifier................. Area of 00000000H-007FFFFFH and 0FF800000H-0FFFFFFFFH

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, NC100 interprets their attributes as follows:

- Variables------------------------------------near attribute
- const-qualified constants-----------------far attribute
- Functions-----------------------------------far attribute

Furthermore, NC100 allows you to modify these default attributes by using the startup options of compile driver nc100.

B.1.2 Format of Variable Declaration

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

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

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

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

```
short near  in_data;
short far   if_data;

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

Figure B.2 Example of Variable Declaration

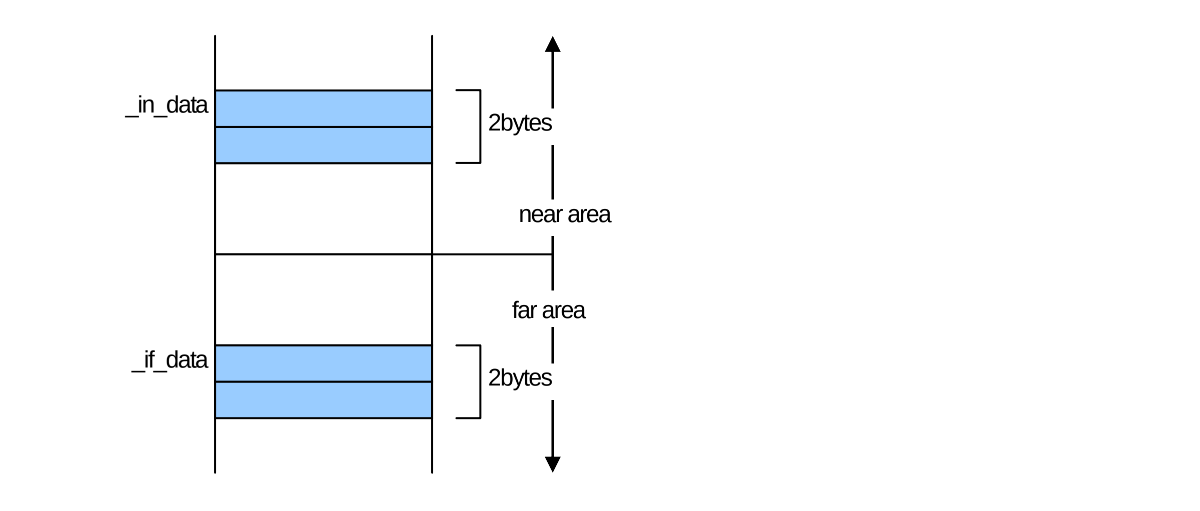


Figure B.3 Memory Location of Variable

B.1.3 Format of Pointer type Variable

The pointer-type variables are always a far-type (4-bytes) variable. If type near is specified in the declaration of a pointer-type variable, the compiler outputs a warning message "Near pointer not supported, near qualifier ignored" and ignores the near qualifier.
An example declaration of a pointer-type variable is shown in Figure B.4.

Example :

```
short       * ptr;
```

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

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

Example :

```
short       far * near   ptr;
```

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

The variable ptr is a 4-byte variable that indicates the short-type variable located in the far area. The ptr itself is located in the near area.
Memory mapping for the above example is shown in Figure B.6.

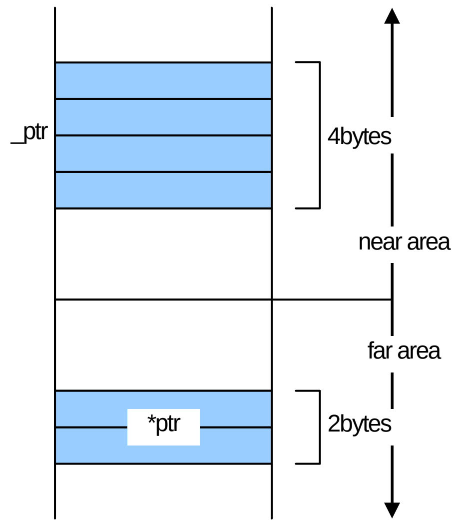


Figure B.6 Memory Location of Pointer type Variable

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

```
Example 1 :
        short     far *     ptr1;
Example 1 :
        short     * far     ptr2;
```

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

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

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

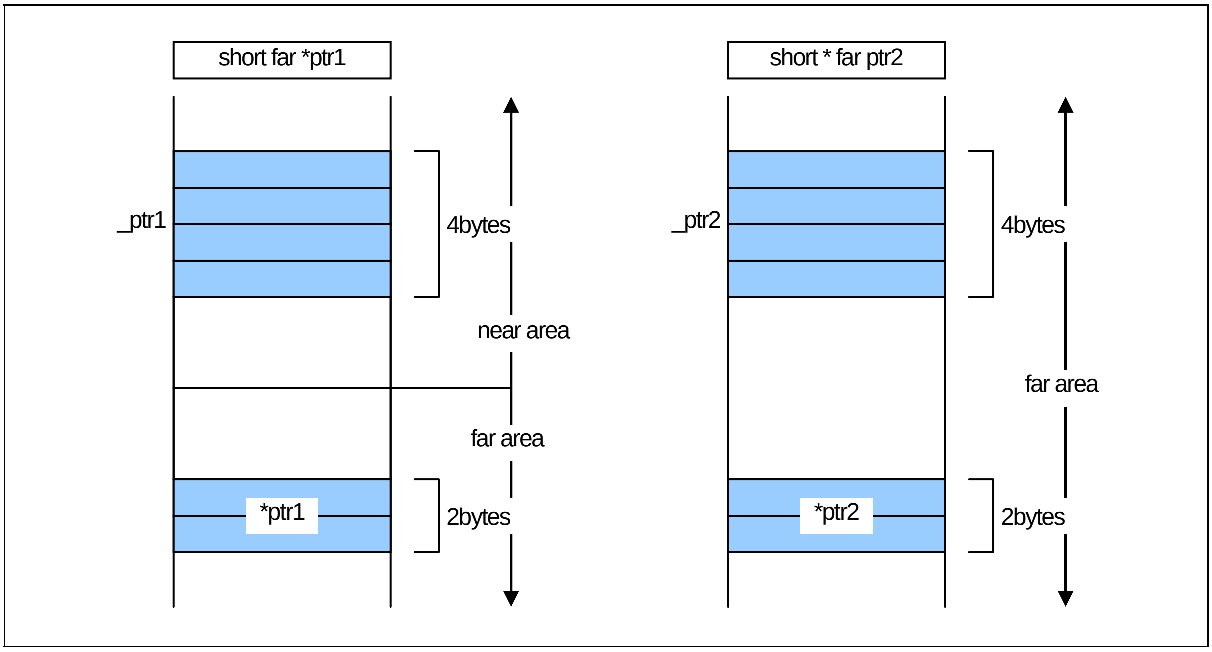


Figure B.9 Memory Location of Pointer type Variable

B.1.4 Declaration of function

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

B.1.5 near and far Control by nc100 Command Line Options

NC100 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. NC100's command line options allow you to modify the default attributes of functions and variables (data). These are listed in the table below.

Table B.3 Command Line Options

Command Line Options	Function
-fnear_ROM(-fNROM)	Change the default attribute of ROM data to near.
-ffar_RAM(-fFRAM)	Change the default attribute of RAM data to far.

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_var;

void        main(void)
{
            f_ptr = &n_var;          /* assigns the near address to the far pointer */
                :
            (abbreviated)
                :
            func ( &n_var );         /* prototype declaration for function with far pointer to parameter */
                                     /* specifies near address parameter at the function call */
}
```

Figure B.10 Type conversion from near to far

When converted to type far, the pointer is sign-extended with the most significant bit of the near address (16-bit quantity).

B.1.7 Declaration of function

In NC100, 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, NC100 outputs a warning and continues processing by assuming the attribute of that function is far. Figure B.11 shows a display example where a function is declared to be near.

```
%nc100   -S   smp.c
R32C/100 Series C Compiler V.X.XX Release XX
Copyright(C) XXXX(XXXX-XXXX). Renesas Technology Corp.
and Renesas Solutions Corp., All rights reserved.
smp.c
[Warning(ccom):smp.c,line 3] function must be far
===> {
func
%
```

Figure B.11 Example Declaration of Function

B.1.8 Function for Specifying near and far in Multiple Declarations

As shown in Figure B.12, if there are multiple declarations of the same variable, the type information for the variable is interpreted as indicating a combined type.

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

void         func(void)
{
             (remainder omitted)
                 :


This Declaration is interpreted as the following:

extern int   far idata = 10;

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

Figure B.12 Integrated Function of Variable Declaration

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

common header file
common.h

```
extern int   far data;
```

C source file
a.c

```
#include "common.h"

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

C source file
b.c

```
#include "common.h"


int          data = 10;
```

Figure B.13 Example of Common header file Declaration

B.1.9 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. NC100 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.14 Example of Variable Declaration

B.2 asm Function

NC100 allows you to include assembly language routines (asm functions)[1] in your C source programs.

B.2.1 Overview of asm Function

The asm function is used for including assembly language code in a C source program. As shown in Figure B.15, the format of the asm function is asm(" ");, where an assembly language instruction that conforms to the AS100 language specifications is included between the double quote marks.

```
#pragma    ADDRESS    ta0_int    6CH
char          ta0_int;

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

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

Figure B.15 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.16.

```
asm( );
```

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

The asm function used in NC100 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.17 below.

```
asm( "     op-code           R1 , $$ [ FB ] " , variable name );
```

Figure B.17 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(void)
{
            short       idata;
            short       a[3];
            struct TAG{
                        short       i;
                        short       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.18 Description example for specifying

Figure B.19 shows an example for referencing an auto variable and its compile result.

```
C source file :

void        func(void)
{
            short idata = 1;

            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 -4
;## #         FRAME     AUTO      (  __PAD2)          size   1,    offset -1
;## #         FRAME     AUTO      (  __PAD1)          size   1,    offset -2
;## #         ARG Size(4)         Auto Size(4)        Context Size(8)
                 :
              (abbreviated)
;## # C_SRC :       asm(" MOV.W $$[FB], R0", idata);
;#### ASM START
MOV.W         -4[FB], R0
              ._line     5
;## # C_SRC :       asm(" CMP.W #00001H ,R0");
              CMP.W      #00001H ,R0
;#### ASM END
              (remainder omitted)
                    :
```

Figure B.19 Example for Referencing an auto Variables

You can also use the format show in Figure B.20 so that auto variables in an asm function use a 1-bit field. (Can not operate bit-fields og greater than 2-bits.)

```
asm( "      op-code    $b[ FB ]" , bit field name);
```

Figure B.20 Format for Specifying FB Offset Bit Position.

You can only specify one variable name using this format. Figure B.21 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.21 Example for Specifying FB Offset Position

Figure B.22 shows examples of referencing auto area bit fields and the result of compiling.

C source file :

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

Assembly language source file(compile result):

```
;## #       FUNCTION func
;## #       FRAME   AUTO    (  __PAD3)       size  1,   offset -1
;## #       FRAME   AUTO    (  __PAD2)       size  1,   offset -2
;## #       FRAME   AUTO    (  __PAD1)       size  1,   offset -3
;## #       FRAME   AUTO    (       s)       size  1,   offset -4
;## #       ARG Size(4)     Auto Size(4)     Context Size(8)

            .SECTION program,CODE,ALIGN
            ._file      'bit.c'
            .align
            ._line      2
;## # C_SRC :           {
            .glb        _func
_func:
            enter       #04H
            ._line      9
;## # C_SRC :           asm(" bset $b[FB]",s.bit1);
;#### ASM START
bset        1,-4[FB]    ;   s
;#### ASM END
            ._line      10
;## # C_SRC :           }
            exitd
```

Figure B.22 Example of Referencing auto Area Bit Field

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.23 below[1]

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

Figure B.23 Description Format for Register Variables

You can only specify two variable name using this format. Figure B.24 shows examples of referencing register variables and the results of compiling.

[1] If the variables need to be forcibly mapped to registers using the register qualifier, specify the option -fenable_register (-fER) when compiling.

C Source file :

```
void        func(void)
{
            register short        i=1;

            asm("      mov.w      $$,R1",i);
}
```

Assembly language source file (compile result) :

```
;## #       FUNCTION func
;## #       ARG Size(4)        Auto Size(0)        Context Size(4)

            .SECTION  program,CODE,ALIGN
            ._file    'reg.c'
            .align
            ._line    2
;## # C_SRC :         {
            .glb      _func
_func:
            ._line    3
;## # C_SRC :         register short i=1;
            mov.w     #0001H,R0 ;   i
            ._line    4
;## # C_SRC :         asm(" mov.w $$,R1",i);
;#### ASM START
           mov.w      R0,R1                      ← R0 register is transferred to R0 register
;#### ASM END
```

Figure B.24 Example for Referencing a Register Variable

In NC100, 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.

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.25 to use extern and static variables in asm functions.

```
asm( "      op-code       R1, $ " , variable name );
```

Figure B.25 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)

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

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

Figure B.26 Example for Specifying Symbol Names

See Figure B.27 for examples of referencing extern and static variables.

C source file :

```
extern short    ext_val;

void        func(void)
{
            static short    s_val;

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

Assembly language source file(compile result) :

```
_func:
            ._line      5
;## # C_SRC :           asm(" mov.w #01H,$$",ext_val);
;#### ASM START
            mov.w       #01H,_ext_val                   ← Move to_ext_val
            ._line      6
;## # C_SRC :           asm(" mov.w #01H,$$",s_val);
            mov.w       #01H,___S0_s_val                ← Move to__S0_s_val
#### ASM END
            ._line      7
;## # C_SRC :           }
            rts
E1:
            .glb        _ext_val

            .SECTION    bss_NEAR,DATA,ALIGN
___S0_s_val:                        ;### C's name is s_val
            .blkb       2
            .END
```

Figure B.27 Example of Referencing extern and static Variables

You can use the format shown in Figure B.26 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[FB]", bit field name);
```

Figure B.28 Format for Specifying Symbol Names

You can specify one variable name using this format. See Figure B.29 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.29 Example of Specifying Symbol Bit Position

Figure B.30 shows the results of compiling the C source file shown in Figure B.29.

```
;## #     FUNCTION func
;## #     ARG Size(4)         Auto Size(0)        Context Size(4)

          .SECTION            program,CODE,ALIGN
          ._file    'bitfield.c'
          .align
          ._line    8
;## # C_SRC :       {
          .glb      _func
_func:
          ._line    9
;## # C_SRC :       asm(" bset $b",s.bit1);
;#### ASM START
           bset     1,_s                ← Reference to bitfield bit0 of structure s
;#### ASM END
          ._line    10
;## # C_SRC :       }
          rts
E1:

          .SECTION            bss_NEAR,DATA,ALIGN
          .glb      _s
_s:
          .blkb     1
          .END
```

Figure B.30 Example of Referencing Bit Field of Symbol

B.2.5 Specification Not Dependent on Storage Class

The variables written in C language can be used in the asm function without relying on the storage class of that variable (auto, register[1], extern, or static variable).
Consequently, any variable written in C language can be used in the asm function by writing it in the format shown in Figure B.31[2]

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

Figure B.31 Description Format Not Dependent on Variable's Storage Class

You can only specify one variable name using this format. Figure B.32 shows examples of referencing register variables and the results of compiling.

```
C source file :

extern int  e_val;

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

            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        #04H
            ._line       7
;## # C_SRC :            asm(" mov.w #1, $@", e_val);
;#### ASM START
            mov.w        #1, _e_val:16                   ← Reference to external variable
            ._line       8
;## # C_SRC :            asm(" mov.w #2, $@", f_val);
            mov.w        #2, -4[FB]                      ← Reference to auto variable
            ._line       9
;## # C_SRC :            asm(" mov.w #3, $@", r_val);
            mov.w        #3, R2R0                        ← Reference to register variable
            ._line       10
;## # C_SRC :            asm(" mov.w #4, $@", s_val);
            mov.w        #4, ___S0_s_val:16              ← Reference to static variable
            ._line       11
;## # C_SRC :            asm(" mov.w $@, $@", f_val,r_val);
            mov.w        -4[FB], R2R0
;#### ASM END
```

Figure B.32 Example for Referencing Variables of Each Storage Class

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

B.2.6 Selectively suppressing optimization

In Figure B.33, the dummy asm function is used to selectively suppress a part of optimization.

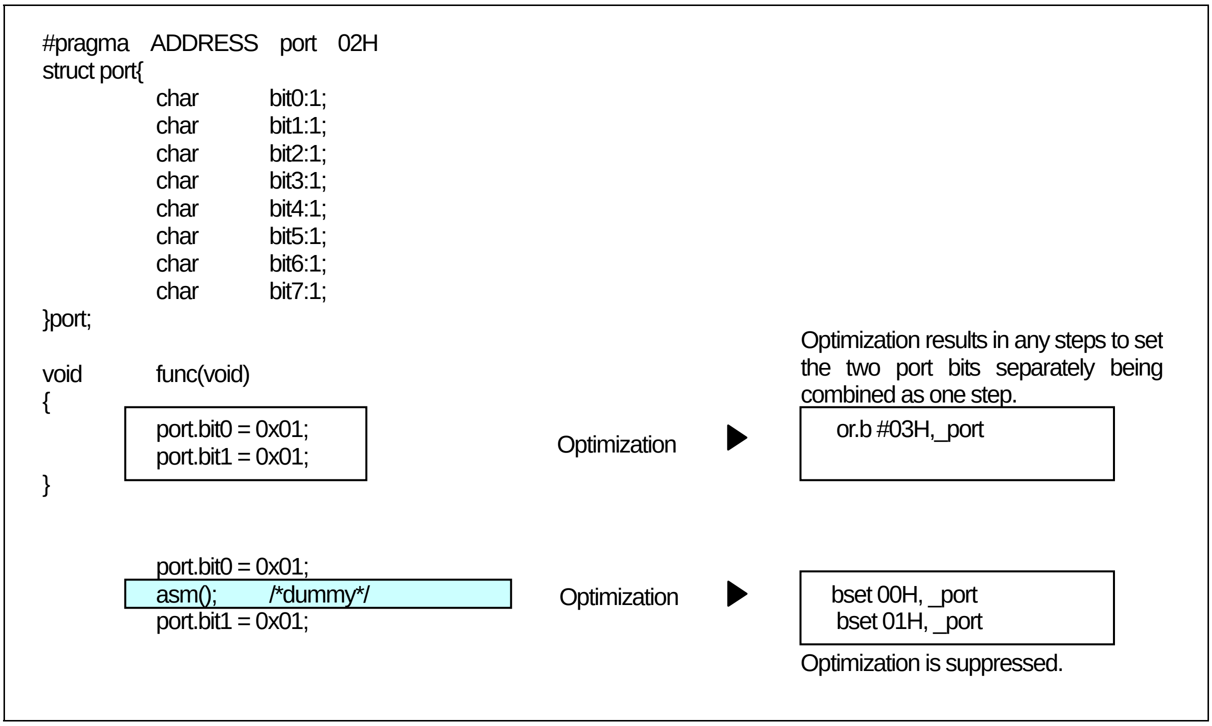


Figure B.33 Example of Suppressing Optimization by Dummy asm

B.2.7 Notes on the asm Function

a Extended Features Concerning asm functions

When using the asm function for the following processing, be sure to use the format shown in the coding examples.

(1) For variables with storage class auto, arguments, and 1-bit bit fields

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.34 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.34 Example Coding of asm Function (1/2)

(2) Specification of the register storage class

You can specify the register storage class in NC100. When register class variables are compiled with option -fenable_register (-fER), use the format shown in Figure B.35 for register variables in asm functions.

```
asm("    MOV.W    #0,$$", i);            ← Format for checking register variables
```

Figure B.35 Example Coding of asm Function (2/2)

Note that, when you specify option -O[1-5], -OR, -OS, -OR_MAX, or -OS_MAX parameters passed via the registers may, to improve code efficiency, be processed as register variables rather than being moved to the auto area. In this case, when parameters are specified in an asm function, the assembly language is output using the register names instead of the variable's FB offset.

(3) When referencing arguments in the asm function

The compiler analyzes a program flow with respect to its interval in which variables (including arguments and auto variables) remain effective as it processes the program. If arguments or auto variables are referenced in an asm function as shown in Figure B.36, the compiler will fail to keep track of the effective interval and cannot generate correct code.
Therefore, if arguments or auto variables need to be referenced in an asm function you write, always be sure to use the “$$, $b, or $@” feature of the asm function for that reference.

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

Figure B.36 Example cannot be referred to correctly

In the above case, because the compiler determines that "i" and "j" are not used within the function func, it does not output codes necessary to construct the frame in which to reference the arguments. For this reason, the arguments cannot be referenced correctly.

(4) About branching within the asm function

The compiler analyzes program flow in the intervals in which registers and variables respectively are effective, as it processes the program. Do not write statements for branching (including conditional branching) in the asm function that may affect the program flow.

b About Register

- Do not destroy registers within the asm function. If registers are going to be destroyed, use push and pop instructions to save and restore the registers.
- NC100 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]");
             :
          (omitted
             :
asm("     .SB       __SB__);
asm("     LDC       #__SB__,SB");           ←SB returned to original state
```

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

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

c Notes on Labels

The assembler source files generated by NC100 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 :

          A1:
          C9830:

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

          __LABEL:
          ___START:
```

Figure B.38 Label Format Prohibited in asm Function

B.3 Description of Japanese Characters

NC100 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. NC100 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 NC100.

- EUC (excluding user-defined characters made up of 3-byte code)
- Shift JIS (SJIS)

B.3.2 Settings Required for Using Japanese Characters

The following environment variables must be set in order to use kanji codes. default specifies :

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

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

```
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 NC100, the input kanji codes are processed by the cpp100 preprocessor. cpp100 changes the codes to EUC codes. In the last stage of token analysis in the ccom100 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 L"漢字文字列" as with normal character strings, it is processed as a pointer type to a char type when manipulating the character string. You therefore cannot manipulate them as 2-byte characters.
To process the Japanese as 2-byte characters, precede the character string with L and process it as a pointer type to a wchar_t type. wchar_t types are defined (typedef) as unsigned short types in the standard header file stdlib.h.
Figure B.41 shows an example of a Japanese character string.

```
#include <stdlib.h>

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

            (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.

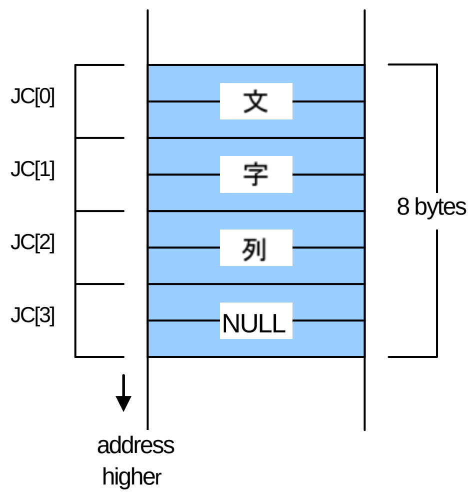


Figure B.42 Memory Location of wchar_t Type Character Strings

B.3.4 Sing 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 ' 文字 ', in 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        func(void)
{
            wchar_t     JC[5];

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

            (remainder omitted)
                :
```

Figure B.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.

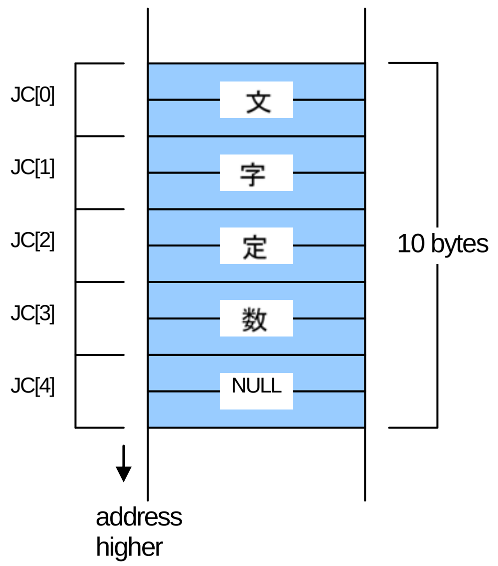


Figure B.45 Memory Location of wchar_t Type Character Constant Assigned Array

B.4 Default Argument Declaration of Function

NC100 allows you to define default values for the arguments of functions in the same way as with the C++ facility. This chapter describes NC100's facility to declare the default arguments of functions.

B.4.1 Overview of Default Argument Declaration of Function

NC100 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 specifier△Type declarator△Declarator([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.

```
short   func( short i=1 , short j=2 );    ← Declares the default values of parameters in the arguments to
                                            the function func as first argument: 1 and second argument: 2.

void    main(void)
{
        func();        ← The actual argument consists of the first argument: 1 and the second argument: 2.
        func(3);       ← The actual argument consists of the first argument: 3 and the second argument: 2.
        func(3,5);     ← The actual argument consists of the first argument: 3 and the second argument: 5.
}
```

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

```
;## # C_SRC :          {
            .glb       _main
_main:
            ._line     5
;## # C_SRC :                     func();
            mov.w      #0002H,R1                    ← second argument :2
            mov.w      #0001H,R0                    ← first argument   :1
            jsr        $func
            ._line     6
;## # C_SRC :                     func(3);
            mov.w      #0002H,R1                    ← second argument :2
            mov.w      #0003H,R0                    ← first argument   :3
            jsr        $func
            ._line     7
;## # C_SRC :                     func(3,5);
            mov.w      #0005H,R1                    ← second argument :5
            mov.w      #0003H,R0                    ← first argument   :3
            jsr        $func
            ._line     8
;## # C_SRC :          }
            rts
               :
            (omitted)
               :

Note) In NC100, 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.

```
short       near sym ;
short       func( short 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:16,R0          ← 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.

a When specifying a default value for multiple arguments

When specifying a default value in a function that has multiple arguments, always be sure to write values beginning with the last argument. Figure B.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

b When specifying a variable for a default value

When specifying a variable for a default value, write the prototype declaration of a function after declaring the variable you specify. If a variable is specified for the default value of an argument that is not declared before the prototype declaration of a function, it is processes as an error.

B.5 inline Function Declaration

NC100 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.

```
inline△type specifier△function;
```

Figure B.52 Declaration Format of inline Storage Class

An example function declaration and its compile result are shown in Figure B.53 and Figure B.54, respectively.

```
inline short  func(short i)          ← Inline function declaration and definition
{
              return i++;
}

void          main(void)
{
              short       s;

              s = func(s);           ← Inline function call
}
```

Figure B.53 Sample program of inline function (sample. c)

```
            .SECTION  program,CODE,ALIGN
            ._file    'sample.c'
            .align
            ._line    7
;## # C_SRC :         {
            .glb      _main
_main:
            enter     #04H
            ._line    10
;## # C_SRC :                    s = func(s);
            mov.w     -4[FB],R0  ;  s
            ._line    2
;## # C_SRC :         {
            mov.w     R0,-2[FB]  ;  i
            ._line    3
;## # C_SRC :                    return i++;        ← Inline storage class have codes
            mov.w     R0,R1                             embedded directly
            add.w     #0001H,R0
            ._line    10
;## # C_SRC :                    s = func(s);
            mov.w     R1,-4[FB]  ;  s
            ._line    11
;## # C_SRC :         }
            exitd
E1:
            .END
```

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 parameter of inline functions

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

(2) Regarding the indirect call of inline functions

The indirect call of an in line function cannot be carried out. It becomes a compile error when a indirect call is described.

(3) Regarding the recursive call of inline functions

The recursive call of an in line function cannot be carried out. It becomes a compile error when a recursive call is described.

(4) Regarding the definition of an inline function

When specifying inline storage class for a function, be sure to define the body of the function 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 NC100.

```
inline void   func(int i);

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

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

Figure B.55 Example of inappropriate code of inline function (1/4)

Furthermore, if any function is defined as an inline function after being used as an ordinary function, the specification of inline has no effect and all of such a definition is handled as static functions (Figure B.56).

```
int           func(int i);

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

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

[Warning Message]
[Warning(ccom):smp.c,line 10] inline function is called as normal function before,change to static function.

Figure B.56 Example of inappropriate code of inline function (2/4)

(5) Regarding the address of an inline function

The inline function itself does not have an address. Therefore, if the & operator is used for an inline function, the software assumes an error. (Figure B.57)

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

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

                f = &func;
}
```

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

Figure B.57 Example of inappropriate code of inline function (3/4)

(6) Declaration of static data

If static data is declared in an inline function, the body of the declared static data is allocated in units of files. For this reason, if an inline function consists of two or more files, this results in accessing different areas. Therefore, if there is static data you want to be used in an inline function, declare it outside the function. If a static declaration is found in an inline function, NC100 generates a warning. Renesas does not recommend entering static declarations in an inline function. (Figure B.58)

```
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.58 Example of inappropriate code of inline function (4/4)

(7) Regarding debug information

NC100 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

NC100 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.59 shows comment format.

```
// comments
```

Figure B.59 Comment Format

Figure B.60 shows example comments.

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

Figure B.60 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.4 Memory Mapping Extended Functions (1/2)

Extented function	Description

#pragma EXTMEM	Declares that data or function be located in an area that cannot be accessed in address-0 relative addressing.
	Syntax :
	`#pragma EXTMEM△variable-name`
	`#pragma EXTMEM△function-name()`
	Example :
	`#pragma EXTMEM val`
	`#pragma EXTMEM func()`

#pragma MONITORn	Declares that data be located in a special section for the RAM monitor.
	Syntax :
	`#pragma MONITOR1△variable-name`
	Example :
	`#pragma MONITOR1 val`

#pragma ROM	Maps the specified variable to rom.
	Syntax :
	`#pragma ROM△variable-name`
	Example :
	`#pragma ROM val`

#pragma SB16DATA	Declares that the data uses SB relative addressing of 16-bit displacement.
	Syntax :
	`#pragma SB16DATA△variable-name`
	Example :
	`#pragma SB16DATA val`

#pragma SBDATA	Declares that the data uses SB relative addressing of 8bit displacement
	Syntax :
	`#pragma SBDATA△variable-name`
	Example :
	`#pragma SBDATA val`

#pragma SECTION	Changes the section name generated by NC100.
	Syntax :
	`#pragma SECTION△section-name△new-section-name`
	Example :
	`#pragma SECTION bss nonval-data`

Table B.5 Memory Mapping Extended Functions (2/2)

Extented function	Description

#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`

b Using Extended Functions for Target Devices

Table B.6 Extended Functions for Use with Target Devices (1/2)

Extended function	Description

#pragma ADDRESS	Specifies the absolute address of a variable. For near variables, this specifies the address within the bank.
	Syntax :
	`#pragma ADDRESS△variable-name△absolute-address`
	Example :
	`#pragma ADDRESS port0 2H`

#pragma DMAC	Specifies the DMAC register of a external variable.
	Syntax :
	`#pragma DMAC△variable-name△DMAC register-name`
	Example :
	`#pragma DMAC dsa0 DSA0`

#pragma INTCALL	Declares a function written in assembler called in a software interrupt (int instruction).
	Syntax1 :
	`#pragma INTCALL△INT-No.△assembler function-name(register-name)`
	Example1 :
	`#pragma INTCALL 25 func(R0,R1)`
	Syntax2 :
	`#pragma INTCALL△INT-No.△C language function-name()`
	Example2 :
	`#pragma INTCALL 25 func()`
	● Always be sure to declare the prototype of the function before entering this declaration.

Table B.7 Extended Functions for Use with Target Devices (2/2)

<table>
<tr><th>Extended function</th><th>Description</th></tr>
<tr><td>#pragma INTERRUPT</td><td>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.
Syntax：
<code>#pragma INTERRUPT△[/B¦/E¦/F¦/R¦/V]△interrupt-handling-function-name</code>

<code>#pragma INTERRUPT△[/B¦/E¦/F¦/R]△interrupt-vector-number△interrupt-handling-function-name</code>

<code>#pragma INTERRUPT△[/B¦/E¦/F¦/R]△interrupt-handling-function-name(vect=interrupt-vector-number)</code>

Example：
<code>#pragma INTERRUPT int_func</code>
<code>#pragma INTERRUPT /B int_func</code>
<code>#pragma INTERRUPT 10 int_func</code>
<code>#pragma INTERRUPT /E 10 int_func</code>
<code>#pragma INTERRUPT int_func(vect=10)</code>
<code>#pragma INTERRUPT /R int_func</code></td></tr>
<tr><td>#pragma PARAMETER</td><td>Declares that, when calling an assembler function, the parameters are passed via specified registers.
Syntax：
<code>#pragma PARAMETER△function-name(register-name)</code>
Example：
<code>#pragma PARAMETER asm_func(R0,R1)</code>
● Always be sure to declare the prototype of the function before entering this declaration.</td></tr>
</table>

c The Other Extensions

Table B.8 The Other Extensions

Extended feature	Description

#pragma __ASMMACRO	Declares defined a function by assembler macro.
	Syntax :
	`#pragma __ASMMACRO△function-name(register-name)`
	Example :
	`#pragma __ASMMACRO max_w(R0,R2)`

#pragma ASM
#pragma ENDASM
	Specifies an area in which statements are written in assembly language.
	Syntax :
	`#pragma△ASM`
	`#pragma△ENDASM`
	Example :
	`#pragma ASM`
	`mov.w R0,R1`
	`add.w 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

NC100 includes the following memory mapping extended functions.

#pragma EXTMEM

Declares exclusion of address-0 relative addressing

Function: Declares a variable or function to be located in an area that cannot be accessed in address-0 relative addressing.

Syntax:

```
#pragma EXTMEM△variable-name
#pragma EXTMEM△function-name()
```

Description: In address-0 relative addressing that is normally used to access an external variable directly or call a function directly, no addresses in the range 00800000H-0FF7FFFFFH can be accessed.
#pragma EXTMEM declares that a variable or function be located in such an inaccessible area. The variables and functions declared with #pragma EXTMEM are accessed in address register relative addressing, etc.
Note that the variables and functions declared with #pragma EXTMEM too can have their location addresses handled by an ordinary far pointer.

Rules:

(1) Declaration with #pragma EXTMEM is ignored unless it declares a variable name or function name.
(2) #pragma EXTMEM is not applied to the static variables declared within a function.
(3) This extended feature has priority over the near and far qualifiers declared.
(4) This extended feature cannot be used in combination with other extended features of #pragma for one variable or function at a time.

Example:

```
#pragma  EXTMEM  extfunc()
#pragma  EXTMEM  extvar;

short  extfunc( void );
short  far  extvar;
short  far  *p;

void        func( void )
{
            extvar  =  extfunc();
            p  =  &extvar;
}
```

Figure B.61 Example Use of #pragma EXTMEM Declaration

Supplement: When this extended feature is specified, the compiler generates the following sections.

Name	Attribute	Content
data_EXT	DATA	Data with initial values declared with #pragma EXTMEM
bss_EXT	DATA	Data without initial values declared with #pragma EXTMEM
rom_EXT	ROMDATA	const qualified data declared with #pragma EXTMEM
data_EXT_INIT	ROMDATA	Initial value of data_EXT section
program_EXT	CODE	Function code declared with #pragma EXTMEM

#pragma MONITORn

Directive to specify the location of the RAM monitor area

Function: Declares that the specified external variable be located in a section used exclusively for the RAM monitor area.

Syntax: `#pragma MONITOR[n]△external variable-name`
(n=1-4)

Rules:

(1) Only external variables and external static variables can be specified.

(2) The area for the external variable declared by #pragma MONITOR[n] is allocated to one of the sections listed below.

- data_MON[n] .... External variables that have initial values are located here
- bss_MON[n] .... External variables that do not have initial values are located here
- data_MON[n]_INIT .... Initial values of external variables that have initial values are located here

(3) The declaration of #pragma MONITOR[n] must be made before the external variable is defined.

(4) The external variable declared by #pragma MONITOR[n] cannot be used in combination with other extended #pragma directives. However, if #pragma SBDATA and #pragma MONITOR[n] are specified at the same time, #pragma SBDATA has priority. At this time, no warnings are output.

Note:

(1) #pragma MONITOR[n] does not affect the op-codes generated by the compiler. Please pay attention to the near/far attributes of variables.

(2) Even if external variables with different near/far attributes coexist in a section used exclusively for the RAM monitor area, no errors and warnings are assumed. Please pay attention to the near/far attributes of variables.

(3) The sections used exclusively for the RAM monitor area are not subject to size limitations.

(4) The location address of the section allocated by #pragma MONITOR[n] and a process to set the initial value for the external variable should be written in the startup program.

(5) If #pragma MONITOR[n] is declared a number times for one and the same external variable, the #pragma MONITOR[n] declared first is effective.

(6) The external variables declared by #pragma MONITOR[n] are not affected by #pragma SECTION.

(7) The declaration of #pragma MONITOR[n] has no effect if 'n' in it is other than 1-4. If the compile option -Wunknown_pragma[-WUP] or -Wall is specified, a warning is output.

(8) External variables with ROM attribute cannot be handled by #pragma MONITOR[n]. However, if the compile option -fconst_not_ROM[-fCNR] is specified, these variables can be handled by #pragma MONITOR[n].

```
#pramga   MONITOR1          i
const     int               i;   <==== Has no effect
```

(9) Even when variable locations are changed by this function, the addressing mode of generated code is not changed. If the locations of variables that are to be stored in a near-attribute RAM area (addresses 00000000H to 00007FFFH) are changed, the section that contains those variables must be located in a near area (except for the ROMDATA attribute sections of initial values).

#pragma MONITORn

Directive to specify the location of the RAM monitor area

Example:

```
#pragma   MONITOR1   i
#pragma   MONITOR1   c

short   i;
short   j = 0x0100;
```

Figure B.62 Example Use of #pragma MONITORn Declaration

#pragma ROM

Map to rom section

Function: Maps specified data (variable) to rom section

Syntax: `#pragma ROM△variable-name`

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

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

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

Example:

```
C language source program :
#pragma   ROM   i
unsigned short          i;                      ← Variable i, which satisfies condition[1]

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


Assembly language source program :

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

Figure B.63 Example Use of #pragma ROM Declaration

#pragma SB16DATA

SB Relative Addressing Using of 16bit displacement Variable Description Function

Function: Declares that the data uses SB relative addressing of 16bit displacement.

Syntax: `#pragma SBDATA△valuable-name`

Description: The R32C/100 series allows you to choose instructions that can be executed efficiently by using SB relative addressing.
Section accessed by SB relative addressing When it has arranged to the far area,#pragma SB16DATA declares that SB relative addressing of 16bit displacement can be used for the variable when referencing data. This facility helps to generate ROM efficient code.

Rules:

(1) Section accessed by SB relative addressing when using #pragma SB16DATA It is necessary to arrange to a far domain. Therefore, it is necessary to change specification of the section arrangement by the start-up file. For details of how to modify the startup file, see Chapter 2.2.2 "Customizing the Startup Program" and Chapter 2.2.3 “2.2.3 Customizing for NC100 Memory Mapping” in the Operation part of the NC100 User's Manual.

(2) As opposed to the same variable #pragma SBDATA #pragma SB16DATA cannot be specified simultaneously.

(3) If #pragma SB16DATA is specified for anything other than a variable, it is ignored as invalid.

(4) If the specified variable is a static variable declared in a function, the #pragmaSB16DATA declaration is ignored as invalid.

(5) The variable declared to be #pragma SB16DATA is placed in a SB16DATA attribute section when allocating memory for it

(6) If #pragma SB16DATA is declared for ROM data, declaration of #pragma SB16DATA becomes invalid[1]

Example:

```
#pragma   SB16DATA   sym_data
int            far sym_data;

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

Figure B.64 Example Use of #pragma SB16DATA Declaration

Supplement: NC100 is premised on an assumption that the SB register will be initialized after reset and will thereafter be used as a fixed quantity.

[1] Do not write a #pragma SB16DATA declaration for ROM data.

#pragma SBDATA

SB Relative Addressing Using of 8bit displacement Variable Description Function

Function: Declares that the data uses SB relative addressing of 8bit displacement.

Syntax: `#pragma SBDATA△valuable-name`

Description: The R32C/100 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) As opposed to the same variable #pragma SBDATA #pragma SB16DATA cannot be specified simultaneously.

(6) If #pragma SBDATA is declared for ROM data, the data is not placed in a SBDATA attribute section.

Example:

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

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

Figure B.65 Example Use of #pragma SBDATA Declaration

Supplement: NC100 is premised on an assumption that the SB register will be initialized after reset and will thereafter be used as a fixed quantity.

#pragma SECTION

Change section name

Function: Changes the names of sections generated by NC100

Syntax: `#pragma SECTION△section name△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.

- The program, data, rom and bss sections can have their names changed a number of times in one and the same file.
- All other sections cannot have their names changed twice or more.

Example:

```
C source program:

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


Assembly language source program:

;### FUNCTION func
          .section   pro1,CODE,ALIGN              ← Maps to pro1 section
          ._file     'smp.c'
          ._line     9
          .glb       _func
_func:


Change name of data section from data to data1:

#pragma   SECTION   data   data1
int       i;                                      ← Maps to data1_NE section

void      func(void)
{
          (remainder omitted)
}

#pragma   SECTION   data   data2
int       j;                                      ← Maps to data2_NE section

void      sub(void)
{
          (remainder omitted)
}
```

Figure B.66 Example Use of #pragma SECTION Declaration

#pragma SECTION

Change section name

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: String data and const data without initial values are 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:

In NC100, structures are packed. For example, the members of the structure in Figure B.67 are arranged in the order declared without any padding.

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

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

Figure B.67 Example Mapping of Structure Members (1/3)

Rules:

(1) Inhibiting packing

This NC100 extended function allows you to control the mapping of structure members. Figure B.68 is an example of mapping the members of the structure in Figure B.67 using #pragma STRUCT to inhibit packing.

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

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

Figure B.68 Example Mapping of Structure Members (2/3)

As shown Figure B.68, if the total size of the structure members is an odd number of bytes, #pragma STRUCT adds 1 byte as packing after the last member. Therefore, if you use #pragma STRUCT to inhibit padding, all structures have an even byte size.

#pragma STRUCT

Control structure mapping

Rules:

(2) Arranging members

This NC100 extended function allows you to map the all odd-sized structure members first, followed by even-sized members. Figure B.69 shows the offsets when the structure shown in Figure B.68 is arranged using #pragma STRUCT.

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

Member name	Type	Size	Mapped location (offset)
i	short	16 bits	0
j	short	16 bits	2
c	char	8 bits	4

Figure B.69 Example Mapping of Structure Members (3/3)

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

Example:

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

Figure B.70 Example of #pragma STRUCT Declaration

B.7.3 Using Extended Functions for Target Devices

NC100 includes the following extended functions for target devices.

#pragma ADDRESS

Specify absolute address of I/O variable

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

Syntax: `#pragma ADDRESS△variable-name△absolute-address`

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

- Append 'B' or 'b' to binary numbers
- Append 'O' or 'o' to octal numbers
- Write decimal integers only.
- Append 'H' or 'h' to hexadecimal numbers. If the number starts with letters A to F, precede it with 0.

Rules:

(1) All storage classes such as extern and static for variables specified in #pragma ADDRESS are invalid.
(2) Variables specified in #pragma ADDRESS are valid only for variables defined outside the function.
(3) #pragma ADDRESS is valid for previously declared variables.
(4) #pragma ADDRESS is invalid if you specify other than a variable.
(5) No error occurs if a #pragma ADDRESS declaration is duplicated, but the last declared address is valid.
(6) A warning occurs if you include an initialization expression and an initialization expression is invalid.
(7) Normally #pragma ADDRESS operates on I/O variables, so that even though volatile may not actually be specified, the compiler processes them assuming volatile is specified.

Example:

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

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

Figure B.71 #pragma ADDRESS Declaration

#pragma ADDRESS

Specify absolute address of I/O variable

Note: If a variable is used prior to the specification of #pragma ADDRESS as shown in Figure B.72, the specification of #pragma ADDRESS has no effect.

```
char        port;

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

#pragma   ADDRESS   port   100H
```

Figure B.72 Cases where the specification of #pragma ADDRESS has no effect

Supplement: The numeric representation in C language is used to write the absolute address in this declaration form.

#pragma DMAC

Specifies the DMAC register of a external variable

Function: The DMAC register inside CPU is assigned to the specified external variable.

Syntax: `#pragma DMAC△variable-name△DMAC Register Name`

Rules:

(1) You have to declare the variable specified to be #pragma DMAC before description of #pragma DMAC.

(2) It can be specified as #pragma DMAC. #pragma DMAC register name and the type of a variable are as follows.

Register Name	DMD0 DMD1 DMD2 DMD3	DCT0 DCT1 DCT2 DCT3	DCR0 DCR1 DCR2 DCR3	DDA0 DDA1 DDA2 DDA3	DDR0 DDR1 DDR2 DDR3	DSA0 DSA1 DSA2 DSA3	DSR0 DSR1 DSR2 DSR3
Variable Type	unsigned long			To arbitrary models far pointer, However, the pointer to a function cannot be used.			

(3) Two or more #pragma DMAC cannot be declared to the same register.

(4) The "&"(address operator), "()"(function call operator),"[]"(subscript operator), and "->"(indirection operator) cannot be specified to the variable specified by #pragma DMAC.

(5) The variable specified by #pragma DMAC is processed as that to which volatile specification is carried out, even if there is no volatile specification.

Example:

```
void        _far *dda0;
#pragma  DMAC  dda0  DDA0

void        func(void)
{
            unsigned char           buff[10];

            dda0 = buff;
}
```

Figure B.73 #pragma DMAC Declaration

#pragma INTCALL

Declare a function called by the INT instruction

Function: Declares a function called by a software interrupt (by the int instruction)

Syntax:
(1) `#pragmaINTCALL△INT-No.△assembler-function-name(register-name, registername,…)`
(2) `#pragma INTCALL△INT-No.△C-function-name()`

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

Rules:

- Declaring assembler functions
  (1) Before a #pragma INTCALL declaration, be sure to include an assembler function prototype declaration. If there is no prototype declaration, a warning is output and the #pragma INTCALL declaration is ignored.
  (2) Observe the following in the prototype declaration:
    (1) Make sure that the number of parameters in the prototype declaration matches those in the #pragma INTCALL declaration.
    (2) You cannot declare the following types in the parameters in the assembler function:
      - structure types
      - union types
    (3) 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:
    - double types, long types (64-bit registers)
      R3R1R2R0, R7R5R6R4, A1A0, A3A2
    - float types, long types, int types, far*{far pointer}(32-bit registers)
      R2R0, R3R1, R6R4, R7R5, A0, A1, A2, A3
    - short types, int types("-fint_16" option use)(16-bit registers)
      R0, R1, R2, R3, R4, R5, R6, R7
    - char types, _Bool types (8-bit registers)
      R0L, R0H, R1L, R1H, R2L, R2H, R3L, R3H
    - There is no differentiation between uppercase and lowercase letters in register names.
  (4) You can only use decimals for the INT Numbers.
- Declaring functions of which the body is written in C
  (1) Before a #pragma INTCALL declaration, be sure to include a prototype declaration. If there is no prototype declaration, a warning is output and the #pragma INTCALL declaration is ignored.
  (2) You cannot specify register names in the parameters of functions that include the #pragma INTCALL declaration.
  (3) Observe the following in the prototype declaration:
    (1) In the prototype declaration, you can only declare functions in which all parameters are passed via registers, as in the function calling rules.
    (2) You cannot declare the following functions as the return values of functions:
      - Functions that return structures or unions
  (4) You can only use decimals for the INT Numbers.

#pragma INTCALL

Declare a function called the INT instruction

Example:

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

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

           i = 0x7FFD;
           l = 0x007F;

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

Figure B.74 Example of #pragma INTCALL Declaration(asm function) (1/2)

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

void       main(void)
{
           int         i, j;

           i = 0x7FFD;
           j = 0x007F;

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

Figure B.75 Example of #pragma INTCALL Declaration(C language function) (2/2)

Supplement: To use the startup file included with the product, alter the content of the vector section before use. For details on how to alter it, refer to “Preparing the Startup Program.”

#pragma INTERRUPT

Declare interrupt function

Function: Declares an interrupt handler

Syntax:

(1) `#pragma INTERRUPT△[/B|/E|/F|R|/V]△interrupt-handler-name`

(2) `#pragmaINTERRUPT△[/B|/E|/F|/R]△interrupt-vector-number△interrupt-handler-name`

(3) `#pragmaINTERRUPT△[/B|/E|/F|/R]△interrupt-handler-name(vect=interrupt-vector-number)`

Description:

(1) By using the above format to declare interrupt processing functions written in C, NC100 generates the code for performing the following interrupt processing at the entry and exit points of the function.
- In entry processing, all registers of the Micro Procesor are saved to the stack.
- In exit processing, the saved registers are restored and control is returned to the calling function by the REIT instruction.

(2) You may specify either /B or /E of /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]
  :Return to th calling function by the FREIT instruction in exit processing.
- [/R]
  Does not output the code that changes floating-point rounding mode of FLG register to the "nearest value."
- [/V]
  Only generates a vector table for interrupt functions and does not change generated code. Use this switch primarily for fixed vectors.

(3) Interrupt vector numbers can be specified in a function declaration.
A variable vector table can be automatically generated by setting interrupt vector numbers before compiling the sources.
To use the assembly language startup program without specifying vector numbers, refer to paragraph e, "Setting an interrupt vector table," in Section 2.2.2, "Customizing the Startup Program."

#pragma INTERRUPT

Declare interrupt function

Rules:

(1) A warning is output when compiling if you declare interrupt processing functions that take parameters
(2) A warning is output when compiling if you declare interrupt processing functions that return a value. Be sure to declare that any return value of the function has the void type.
(3) Only functions for which the function is defined after a #pragma INTERRUPT declaration are valid.
(4) No processing occurs if you specify other than a function name.
(5) No error occurs if you duplicate #pragma INTERRUPT declarations.
(6) You cannot specify both switch /E and switch /B at the same time.
(7) If different interrupt vector numbers are written in the same interrupt handling function, the vector number declared later is effective.
(8) /V and other switches cannot be used at the same time.

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

Figure B.76 Example for writing different interrupt vector numbers

Example:

```
extern int    int_counter;

#pragma   INTERRUPT   /B   i_func

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

Figure B.77 Example of #pragma INTERRUPT Declaration

Supplement:

(1) To use the startup file included with the product, alter the content of the vector section before use. For details on how to alter it, refer to “Preparing the Startup Program.”
(2) When using a register on the back side, be careful that the back register is not corrupted by a nesting of interrupts.

#pragma PARAMETER

Declare assembler function that passed arguments via register

Function: Declares an assembler function that passes parameters via registers

Syntax: `#pragma PARAMETER△assembler-function-name(register-name,register-name,…)`

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

- double types, long long types (64-bit registers)
  R3R1R2R0, R7R5R7R4, A1A0, A3A2
- float types, long types, int types, far *{far pointer} (32-bit registers)
  R2R0, R3R1, R6R4, R7R5, A0, A1, A2, A3
- short types, int types("-fint_16" option use)(16-bit registers)
  R0, R1, R2, R3, R4, R5, R6, R7
- char types, _Bool types(8-bit registers)
  R0L, R0H, R1L, R1H, R2L, R2H, R3L, R3H
- There is no differentiation between uppercase and lowercase letters in register names.
- Structure and union types cannot be declared.

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

c The assembler functions shown below cannot be declared:
- Functions returning structure or union type

Example:

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

void        main(void)
{
            short       i, j;

            i = 0x7FFD;
            j = 0x007F;

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

Figure B.78 Example of #pragma PARAMETER Declaration

B.7.4 Use of the other extension function

NC100 includes the following extended function for embedding assembler description inline.

#pragma __ASMMACRO

Assembler macro function

Function: Declares defined a function by assembler macro.

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

Rules:

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

(2) Can’t declare the function of no parameter. Parameter is passed via register. Please specify the register matching the parameter type.

(3) Please append the underscore (“_”) to the head of the definition assembler macro name.

(4) The following is a return value-related calling rules. You can’t declare structure and union type as the return value.

char and _Bool types :	R0L	float types :	R2R0
int("-fint_16" use), short types :	R0	double types :	A1A0
int("-fint_16" does'nt use), long types :	R2R0	long-long type :	A1A0
pointer types :	A0		

(5) If you change the register’s data, save the register to the stack in entry processing of assembler macro function and the saved register restore in exit processing.

Example:

```
static short  max_w( short, short );             /* Be sure to declare “static” */

#pragma __ASMMACRO max_w( R0, R2 )
#pragma ASM
_max_w .macro
          max.w R2,R0 ;                          ; The return-value is set to, R0 register
          .endm
#pragma ENDASM

short     s;

void      test_func( void )
{
s         = max_w( 2, 3 );
}
```

Figure B.79 Example of #pragma __AMMACRO

#pragma ASM-#pragma ENDASM

Inline assembling

Function: Specifies assembly code in C.

Syntax:

```
#pragma ASM
assembly statements
#pragma ENDASM
```

Description: The line(s) between #pragma ASM and #pragma ENDASM are output without modifying anything to the generated assembly source file.
Writing #pragma ASM, be sure to use it in combination with #pragma ENDASM. NC100 suspends processing if no #pragma ENDASM is found the corresponding #pragma ASM.

Rules:

(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(void)
{
            int         i, j;

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

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

#pragma   ENDASM
}
```

This area is output directly to an assembly language file.

Figure B.80 Example of #pragma ASM(ENDASM)

Supplement: It is this assembly language program written between #pragma ASM and #pragma ENDASM that is processed by the C preprocessor.

#pragma JSRA

Calls a function with JSR.A

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

Syntax: `#pragma JSRA△function-name`

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

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

Example:

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

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

Figure B.81 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:

```
#pragma   JSRW   func()

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

Figure B.82 Example of #pragma JSRW

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

#pragma PAGE

Output .PAGE

Function: Declares the position to be changed for a new page in a list file that is output by an assembler.

Syntax: `#pragma PAGE`

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

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

Example:

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

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

Figure B.83 Example of #pragma PAGE

B.8 assembler Macro Function

B.8.1 Outline of Assembler Macro Function

NC100 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 form as C language functions, as shown in Figure B.84.
When using the facility of any assembler macro function, be sure to include asmmacro.h.

```
#include   <asmmacro.h>              /* Includes the assembler macro function definition file */
long        l;
char        a[20];
char        b[20];

void        func(void)
{
            l = rmpa_b(0,19,a,b);     /* asm Macro Function(rmpa command) */
}
```

Figure B.84 Description Example of Assembler Macro Function

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

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

ABS

Function : Returns the absolute value of val

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static  signed char     abs_b( signed char val );

/* When calculated in 16 bits */
static  short int       abs_w( short int val );

/* When calculated in 32 bits */
static  long int        abs_l( long int val );
```

MAX

Function : Returns the value val1 or val2 whichever is found larger by comparison.

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static  signed char     max_b( signed char val1, signed char val2 );

/* When calculated in 16 bits */
static  short int       max_w(  short  int  val1,  short  int  val2  );

/* When calculated in 32 bits */
static  long int        max_l( long int val1, long int val2 );
```

MIN

Function : Returns the value val1 or val2 whichever is found smaller by comparison.

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static  signed char    min_b( signed char val1, signed char val2 );

/* When calculated in 16 bits */
static  short int      min_w(  short  int  val1,  short  int  val2  );

/* When calculated in 32 bits */
static  long int       min_l( long int val1, long int val2 );
```

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>

/* When calculated in 8 bits */
static  long long      rmpa_b( long long init, unsigned long int count,
const signed char _far *p1, const signed char _far *p2 );

/* When calculated in 16 bits */
static  long long      rmpa_w( long long init, unsigned long int count,
const short int _far *p1, const short int _far *p2 );

/* When calculated in 32 bits */
static  long long      rmpa_l( long long init, unsigned long int count,
const long int _far *p1, const long int _far *p2 );
```

SIN

Function :

Strings are transferred from a fixed source address that is indicated by p1 to the destination address indicated by p2 as many times as indicated by count in the address-incrementing direction. There is no return value.

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static void sin_b(volatileconstvoid_far*p1,void_far*p2,unsigned
long int count );

/* When calculated in 16 bits */
static void sin_w(volatileconstvoid_far*p1,void_far*p2,unsigned
long int count );

/* When calculated in 32 bits */
static void sin_l(volatileconstvoid_far*p1,void_far*p2,unsigned
long int count );
```

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 :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static void smovb_b( const void _far *p1, void _far *p2, unsigned long
int count );

/* When calculated in 16 bits */
static void smovb_w( const void _far *p1, void _far *p2, unsigned long
int count );

/* When calculated in 32 bits */
static void smovb_l( const voiid _far *p1, void _far *p2, unsigned
long int count );
```

SMOVF

Function :

Strings are transferred from the source address indicated by p1 to the destination address indicated by p2 as many times as indicated by count in the address incrementing direction. There is no return value.

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static void smovf_b( const void _far *p1, void _far *p2, unsigned long int count );

/* When calculated in 16 bits */
static void smovf_w( const void _far *p1, void _far *p2, unsigned long int count );

/* When calculated in 32 bits */
static void smovf_l( const voiid _far *p1, void _far *p2, unsigned long int count );
```

SMOVU

Function :

Strings are transferred from the source address indicated by p1 to the destination address indicated by p2 in the address-incrementing direction until zero is detected. There is no return value

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static void smovu_b( const void _far *p1, void _far *p2 );

/* When calculated in 16 bits */
static void smovu_w( const void _far *p1, void _far *p2 );
```

SOUT

Function : Strings are transferred in the address-incrementing direction from the source address indicated by p1 to the destination address indicated by p2 as many times as indicated by count. There is no return value.

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static void sout_b( const void _far *p1, volatile void _far *p2, unsigned
long int count );

/* When calculated in 16 bits */
static void sout_w( const void _far *p1, volatile void _far *p2, unsigned
long int count );

/* When calculated in 32 bits */
static void sout_l( const void _far *p1, volatile void _far *p2, unsigned
long int count );
```

SSTR

Function : Strings are stored using val as the data to store, p as the address to from val address which to transfer, and count as the number of times to transfer data. There is no return value.

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static void sstr_b( usigned char val, void _far *p, unsigned long int
count );

/* When calculated in 16 bits */
static void sstr_w( usigned short int val, void _far *p, unsigned long
int count );

/* When calculated in 32 bits */
static void sstr_l( usigned long int val, void _far *p, unsigned long
int count );
```

SUNTIL

Function : Searches the file in the address increment direction from the comparison address indicated by from as many times as specified by count until the data that matches val is encountered.

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static void _far *suntil_b( unsigned char val, const void _far *from,
unsigned long int count );

/* When calculated in 16 bits */
static void _far *suntil_w( unsigned short int val, const void _far
*from,unsigned long int count );

/* When calculated in 32 bits */
static void _far *suntil_l( unsigned long iont val, const void _far
*from,unsigned long int count );
```

SWHILE

Function : Searches continually in the address incrementing direction from the comparison address indicated by from as many times as specified by cout until the data that does not match val is encountered.

Syntax :

```
#include <asmmacro.h>

/* When calculated in 8 bits */
static void _far *swhile_b( unsigned char val, const void _far *from,
unsigned long int count );

/* When calculated in 16 bits */
static void _far *swhile_w( unsigned short int val, const void _far
*from,unsigned long int count );

/* When calculated in 32 bits */
static void _far *swhile_l( unsigned long iont val, const void _far
*from,unsigned long int count );
```

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

NC100 is a cross C compiler targeting the R32C/100 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 R32C/100 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 NC100 Performance

This section provides an overview of NC100 performance.

a Test Environment

Table C.1 shows the standard PC environment.

Table C.1 Standard PC Environment

Item	Type of PC	OS Version
PC environment	IBM PC/AT or compatible	Windows XP, Windows Me, Windows 98, Windows 2000, Windows NT 4.0

b C Source File Coding Specifications

Table C.2 shows the specifications for coding NC100 C source files. Note that estimates are provided for items for which actual measurements could not be achieved.

Table C.2 Specifications for Coding C Source Files

Item	Specification
Number of characters per line of source file	512 bytes (characters) including the new line code
Number of lines in source file	65535 max.

c NC100 Specifications

Table C.3 to Table C.4 lists the NC100 specifications. Note that estimates are provided for items for which actual measurements could not be achieved.

Table C.3 NC100 Specifications (1/2)

Item	Specification
Maximum number of files that can be specified in nc100	Depends on amount of available memory
Maximum length of filename	Depends on operating system
Maximum number of macros that can be specified in nc100 command line option -D	Depends on amount of available memory
Maximum number of directories that can be specified in nc100 command line option -I	50max
Maximum number of parameters that can be specified in nc100 command line option -as100	Depends on amount of available memory
Maximum number of parameters that can be specified in nc100 command line option -n100	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)	64K 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

Table C.4 NC100 Specifications (2/2)

Item	Specification
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 numerator	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 NC100 language specifications with the standard language specifications.

C.2.1 Syntax

This section describes the syntactical token elements. In NC100, the following are processed as tokens:

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

a Key Words

NC100 interprets the followings as key words.

Table C.5 Key Words List

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

b Identifiers

Identifiers consist of the following elements:

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

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

c Constants

Constants consists of the followings.

- Integer constants
- Floating point constants
- Character constants

(1) Integer constants

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

Table C.6 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 . signed long long

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

(1) Unsigned constants
Specify unsigned constants by appending the letter U or u after the value. The type is determined from the value in the following order:
- unsigned int . unsigned long . unsigned long long

(2) long-type constants
Specify long-type constants by appending the letter L or l. The type is determined from the value in the following order:
- Octal and hexadecimal: 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.7 Extended Notation List

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

d Character Literals

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

e Operators

NC100 can interpret the operators shown in Table C.8.

Table C.8 Operators List

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

f Punctuators

NC100 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

NC100 supports the following data type.

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

b Qualified Type

NC100 interprets the following as qualified type.

- const
- volatile
- restrict
- near
- far

c Data Type and Size

Table C.9 shows the size corresponding to data type.

Table C.9 Data Type and Bit Size

Type	Existence of sign	Bit size	Range of values
_Bool	No	8	0, 1
char unsigned char	No	8	between 0 and 255
signed char	Yes	8	between -128 and 127
int short signed int signed short	Yes	16	between -32768 and 32767
unsigned int unsigned short	No	16	between 0 and 65535
int long signed int signed long	Yes	32	between -2147483648 and 2147483647
unsigned int unsigned long	No	32	between 0 and 4294967295
long long signed long long	Yes	64	between -9223372036854775808 and 9223372036854775807
unsigned long long	No	64	18446744073709551615
float	Yes	32	between 1.17549435e-38F and 3.40282347e+38F
double long double	Yes	64	between 2.2250738585072014e-308 and 1.7976931348623157e+308
far pointer	No	32	between 0 and 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.
- Type int is handled in 32 bits. However, if the compile option “fI16 (-fint_16)” is specified, int is handled in 16 bits.

C.2.3 Expressions

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

Table C.10 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
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

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

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

C.2.4 Declaration

There are following two types of declaration.:

- Variable Declaration
- Function Declaration

a Variable Declaration

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

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

Figure C.1 Declaration Format of Variable

(1) Storage-class Specifiers

NC100 supports the following storage-class specifiers.

- extern
- static
- typedef
- auto
- register

(2) Type Declarator

NC100 supports the type declarators.

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

(3) Declaration Specifier

Use the format of declaration specifier shown in Figure C.2 in NC100.

```
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

NC100 allows the initial values shown in Figure C.3 in initialization expressions.

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

Figure C.3 Initial Values Specifiable in Initialization Expressions

b Function Declaration

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

```
function declaration (definition) :
        storage-class specifier△type declarator△declaration specifier△main program

function declaration (prototype declaration) :
        storage-class specifier△type declarator△declaration specifier;
```

Figure C.4 Declaration Format of Function

(1) Storage-class Specifier

NC100 supports the following storage-class specifier.

- extern
- static

(2) Type Declarators

NC100 supports the following type declarators.

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

(3) Declaration Specifier

Use the format of declaration specifier shown in Figure C.5 in NC100.

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

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

Figure C.5 Format of Declaration Specifier

(4) Body of the Program

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

```
List of Variable Declaratoropt Compound Statement

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

Figure C.6 Format of Body of the Program

C.2.5 Statement

NC100 supports the following.

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

a Labelled Statement

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

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

Figure C.7 Format of Labelled Statement

b Compound Statement

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

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

Figure C.8 Format of Compound Statement

c Expression / Null Statement

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

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

Figure C.9 Format of Expression and Null Statement

d Selection Statement

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

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

Figure C.10 Format of Selection Statement

e Iteration Statement

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

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

* opt indicates optional items.
```

Figure C.11 Format of Iteration Statement

f Jump statement

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

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

*opt indicates optional items.
```

Figure C.12 Format of Jump Statement

g Assembly Language Statement

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

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

Figure C.13 Format of Assembly Language Statement

C.3 Preprocess Commands

Preprocess commands start with the pound sign (#) and are processed by the cpp100 preprocessor. This chapter provides the specifications of the preprocess commands.

C.3.1 List of Preprocess Commands Available

Table C.12 lists the preprocess commands available in NC100.

Table C.12 List of Preprocess Commands

Command	Function
#assert	Outputs a warning when a constant expression is false.
#define	Defines macros.
#elif	Performs conditional compilation.
#else	Performs conditional compilation.
#endif	Performs conditional compilation.
#error	Outputs messages to the standard output device and terminates processing.
#if	Performs conditional compilation.
#ifdef	Performs conditional compilation.
#ifndef	Performs conditional compilation.
#include	Takes in the specified file.
#line	Specifies file's line numbers.
#pragma	Instructs processing for NC100's extended function.
#undef	Undefines macros.

C.3.2 Preprocess Commands Reference

The NC100 preprocess commands are described in more detail below.

#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(cpp100):x.c, line xx]assertion warning
```

#define

Function: Defines macros.

Format:
(1) `#define△indentifier△lexical string opt.`
(2) `#define△identifier(identifier list opt)△lexical string opt`

Description:
(3) Defines an identifier as macro.
(4) 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 NC100.

```
R32C100
NC100
__INT_16__ ( When compilation option "-fI16(-fint_16)" is used, it is defined. )
__CHAR_SIGNED__ ( When compilation option "-fSC(-fsigned_char)" is used, it is defined. )
```

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

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

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

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

#error

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

Format:
```
#error△character string
```

Description:

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

#if – #elif – #else – #endif

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

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

Description:

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

#ifndef – #elif – #else – #endif

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

Format:

```
#ifndef△identifier
:
#elif△constant expression
:
#else
:
#endif
```

Description:

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

```
#if△!defined△identifier
#if△!defined△(identifier)
```

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

#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 nc100's command line option -I.
Searches <file name> from the directory specified by environment variable
- "INC100" 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 nc100's startup option -I.
  (2) The directory specified by environment variable "INC100"

(3) If the macro-expanded identifier is <file name> or "file name" this command takes in that file from the directory according to rules of search [1] or [2].

- The maximum number of levels of nesting is 40.
- An include error results if the specified file does not exist.

#line

Function: Changes the line number in the file.

Format: `#line△integer△"file name"`

Description:

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

#pragma

Function: Instructs the system to process NC100's extended functions.

Format:

(1) #pragma ROM△variable name
(2) #pragma SBDATA△variable name
(3) #pragma SB16DATA△variable name
(4) #pragma SECTION△predetermined section name△altered section name
(5) #pragma STRUCT△tag name of structure△unpack
(6) #pragma STRUCT△tag name of structure△arrange
(7) #pragma ADDRESS△variable name△absolute address
(8) #pragma BITADDRESS△variable name△bit position, absolute address
(9) #pragma DMAC△variable name△DMAC register name
(10) #pragma INTCALL△int No△assembler function name (register name, register name, ..)
(11) #pragma INTCALL△int No△C language function name()
(12) #pragma INTERRUPT△[/B ¦/E ¦/F ¦/R ¦/V]△interrupt handling vector number△interrupt handling function name
(13) #pragma PARAMETER△assembler function name (register name, register name, ...)
(14) #pragma ASM
(15) #pragma ENDASM
(16) #pragma JSRA△function name
(17) #pragma JSRW△function name
(18) #pragma PAGE
(19) #pragma __ASMMACRO△function name (register name)
(20) #pragma MRCALL△S=stack size△INT number△function code service call name (type of argument...)
(21) #pragma MRPARAMETER△service call name (type of quotation...)
(22) #pragma ALMHANDLER△alarm handler function name
(23) #pragma CYCHANDLER△cyclic handler function name
(24) #pragma INTHANDLER△[/E ¦ /R]△interrupt handler function name
(25) #pragma TASK△task start function name

#pragma

Description:

(1) Facility to arrange in the rom section
(2) Facility to describe variables using SB relative addressing
(3) Facility to describe variables using SB relative 16-bit displacement addressing
(4) Facility to alter the section base name
(5) Facility to control the array of structures
(6) Facility to control the array of structures
(7) Facility to specify absolute addresses for input/output variables
(8) Facility to specify absolute-with bit position addresses for input/output variables
(9) Facility to specify the DMAC register of a external variable.
(10) Facility to declare functions using software interrupts
(11) Facility to declare functions using software interrupts
(12) Facility to write interrupt functions
(13) Facility to declare assembler functions passed via register
(14) Facility to describe inline assembler
(15) Facility to describe inline assembler
(16) Facility to declare functions calling with JSR.A instruction
(17) Facility to declare functions calling with JSR.W instruction
(18) Facility to output .PAGE
(19) Facility to declare Assembler macro function
(20) Facility to declare interface functions of service call of realtime OS for R32C series
(21) Facility to declare interface functions of service call of realtime OS for R32C series
(22) Facility to declare alarm handler functions of realtime OS for R32C series.
(23) Facility to declare cyclic handler functions of realtime OS for R32C series.
(24) Facility to declare kernel interrupt handler functions of realtime OS for R32C series
(25) Facility to declare task start functions of realtime OS for R32C series.

- You can only specify the above 25 processing functions with #pragma. If you specify a character string or identifier other than the above after #pragma, it will be ignored.
- By default, no warning is output if you specify an unsupported #pragma function. Warnings are only output if you specify the nc100 command line option -Wunknown_pragma (-WUP).

#undef

Function: Nullifies an identifier that is defined as macro.

Format: `#undef△identifier`

Description:

- Nullifies an identifier that is defined as macro.
- The following four identifiers are compiler reserved words. Because these identifiers must be permanently valid, do not undefine them with #undef.

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

C.3.3 Predefined Macros

The following macros are predefined in NC100:

- R32C100
- NC100
- __INT_16__ (When compilation option "-fI16(-fint_16)" is used, it is defined.)
- __CHAR_SIGNED__(When compilation option –fSC "-fsigned_char)" is used, it is defined.)

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-NC100 C programs.

```
#ifdef   NC100
#pragma   ADDRESS   port0   2H
#pragma   ADDRESS   port1   3H
#else
#pragma   AD   portA = 0x5F
#pragma   AD   portA = 0x60
#endif
```

Figure C.14 Usage Example of Predefined Macros

Appendix D C Language Specification Rules

This appendix describes the internal structure and mapping of data processed by NC100, the extended rules for signs in operations, etc, and the rules for calling functions and the values returned by functions.

D.1 Internal Representation of Data

D.1.1 Integral Type

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

Table D.1 Data Size of Integral Type

Type	Existence of sign	Bit size	Range of values
_Bool	No	8	0, 1
char unsigned char	No	8	between 0 and 255
signed char	Yes	8	between -128 and 127
int short signed int signed short	Yes	16	between -32768 and 32767
unsigned int unsigned short	No	16	between 0 and 65535
int long signed int signed long	Yes	32	between -2147483648 and 2147483647
unsigned int unsigned long	No	32	between 0 and 4294967295
long long signed long long	Yes	64	between -9223372036854775808 and 9223372036854775807
unsigned long long	No	64	18446744073709551615
float	Yes	32	between 1.17549435e-38F and 3.40282347e+38F
double long double	Yes	64	between 2.2250738585072014e-308 and 1.7976931348623157e+308
near pointer	No	16	between 0 and 0xFFFF
far pointer	No	32	between 0 and 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.
- Type int is handled in 32 bits. However, if the compile option "-fI16 (-fint_16)" is specified, int is handled in 16 bits.

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	between 1.17549435e-38F and 3.40282347e+38F
double long double	Yes	64	between 2.2250738585072014e-308 and 1.7976931348623157e+308

NC100's floating-point format conforms to the format of IEEE (Institute of Electrical and Electronics Engineers) standards. The following shows the single precision and double precision floating-point formats.

(1) Single-precision floating point data format

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

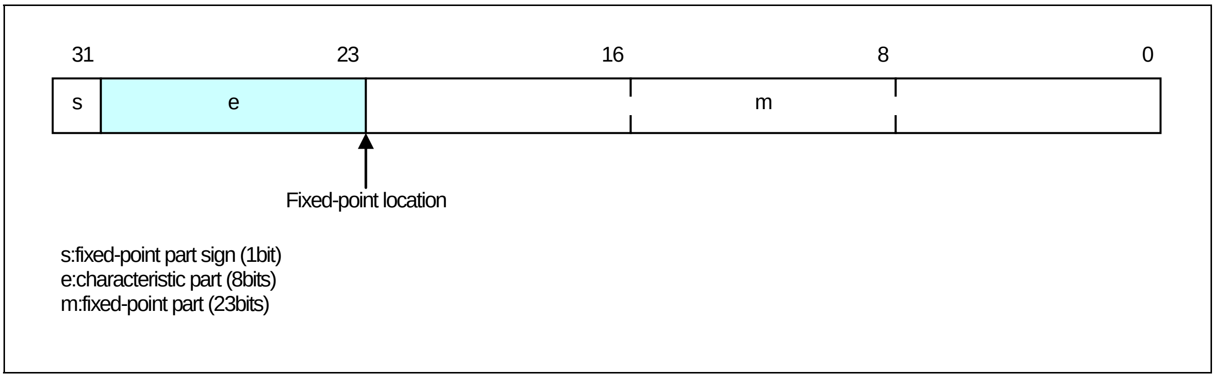


Figure D.1 Single-precision floating point data format

(2) Double-precision floating point data format

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

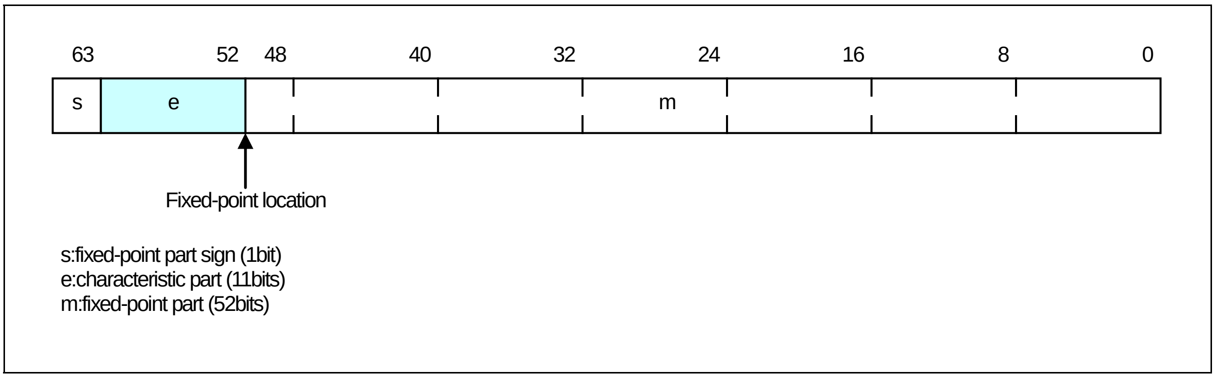


Figure D.2 Double-precision floating point data format

D.1.3 Enumerator Type

Enumerator types have the same internal representation as unsigned int types. Unless otherwise specified, integers 0, 1, 2, are applied in the order in which the members appear.

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
pointers	No	32	between 0 and 0xFFFFFFFF

All pointers are handled as the far pointer. Therefore, the compiler outputs a warning "Near pointer not supported, near qualifier ignored" to the effect that the pointer variables declared as a near pointer will be handled as a far pointer.
Note, however, that if the compile option "-WINP (-Wignore_near_pointer)" is specified, the compiler inhibits said warning from being output.

D.1.5 Array Types

Array types are mapped contiguously to an area equal to the product of the size of the elements (in bytes) and the number of elements. They are mapped to memory in the order in which the elements appear. Figure D.3 is an example of mapping.

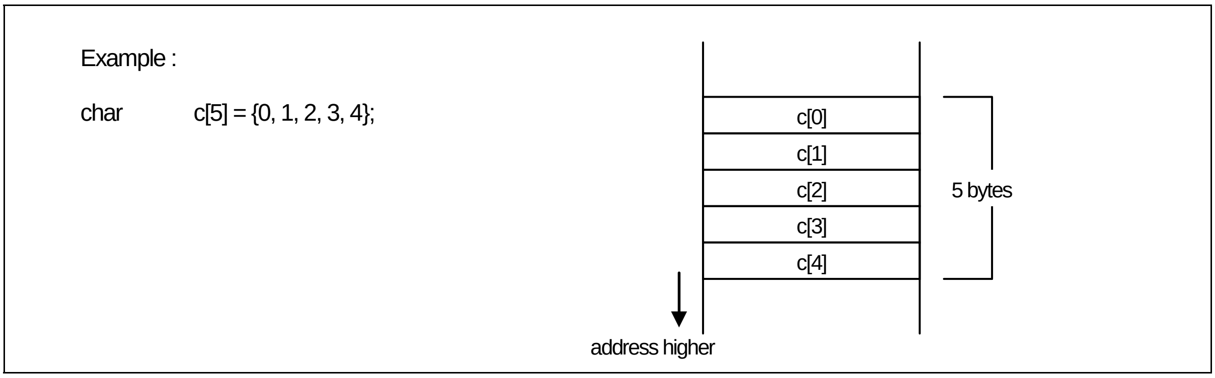


Figure D.3 Example of Placement of Array

D.1.6 Structure types

Structure types are mapped contiguously in the order of their member data. Figure D.4 is an example of mapping.

Example :

```
struct TAG {
        char    c;
        short   i;
} s;
```

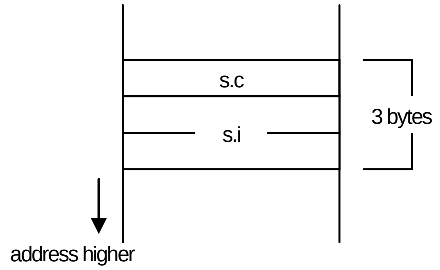


Figure D.4 Example of Placement of Structure (1/2)

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

Example :

```
#pragma STRUCT  TAG  unpac

struct TAG {
        char    c;
        short   i;
} s;
```

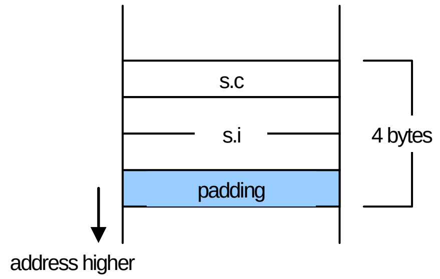


Figure D.5 Example of Placement of Structure (2/2)

D.1.7 Unions

Unions occupy an area equal to the maximum data size of their members. Table D.6 is an example of mapping.

Example :

```
union TAG {
        char    c;
        short   i;
        long    lo;
} s;
```

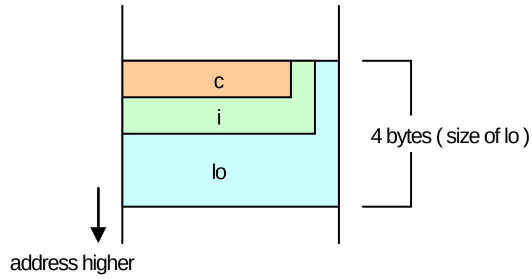


Figure D.6 Example of Placement of Union

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;
```

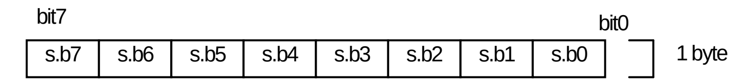


Figure D.7 Example of Placement of Bitfield (1/2)

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;
        short   b56 :2;
        char    b7 : 1;
} s;
```

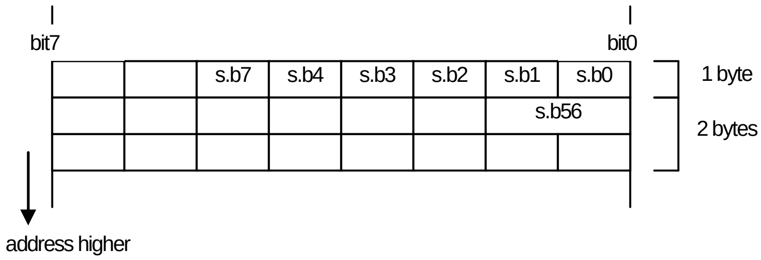


Figure D.8 Example of Placement of Bitfield (2/2)

- Note :
  (1) If no sign is specified, the default bitfield member type is unsigned.
  (2) Can not specifies bit-fields of long long type.

D.2 Sign Extension Rules

Under the ANSI and other standard C language specifications, char type data is sign extended to int type data for calculations, etc. This specification prevents the maximum value for char types being exceeded with unexpected results when performing the char type calculation shown in Figure D.9.

```
void        func(void)
{
            char        c1, c2, c3;

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

Figure D.9 Example of C Program

To generate code that maximizes code efficiency and maximizes speed, NC100 does not, by default, extend char types to int types. The default can, however, be overridden using the nc100 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 NC100 are as follows:
* unsigned char type ....between 0 and 255
* signed char type .... between -128 and 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
char _Bool	R0L register
int (16 bits) short int	R0 register
int (32 bits) float long	R2R0 register
pointer	A0 register
long long double long double	A1A0 register (32 high-order and 32 low-order bits stored in A1 and A0 registers, respectively)
Structure union	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

NC100 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, Table D.6 and Table D.6 to pass arguments.

- Function is prototype declared[1] and the type of argument is known when calling the function.
- Variable argument "..." is not used in prototype declaration.
- For the type of the argument of a function, the Argument and Type of Argument in Table D.5, Table D.6 and Table D.7 are matched.

[1] NC100 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 recommend using a prototype- declaring description format as the standard format to write the C language source files for NC100.

Table D.5 Rules on Argument Transfer via Register (NC100)

Argument	Type of argument	Registers used
First argument	_Bool, char	R0L register
	int (16 bits), short	R0 register
	int (32 bits) float, long	R2R0 register
	pointer	A0 register
	long long, double long double	A1A0 register
Second argument	_Bool, char	R1L register
	int (16 bits), short	R1 register
	int (32 bits) float, long	R3R1 register
	pointer	A2 register
	long long, double long double	A3A2 register
Third argument	int (16 bits), short	R4 register
	int (32 bits) float, long	R6R4 register
	pointer	R6R4 register
Fourth argument	int (16 bits), short	R5 register
	int (32 bits) float, long	R7R5 register
	pointer	R7R5 register

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

Argument	Type of argument	Registers used
First argument	_Bool char	R0L register
	int near pointer	R0 register

Table D.7 Rules on Argument Transfer via Register (NC30)

Argument	Type of argument	Registers used
First argument	_Bool char	R1L register
	int near pointer	R1 register
Second argument	int near pointer	R2 register

(2) Passing arguments via stack

All arguments that do not satisfy the register transfer requirements are passed via stack. The Table D.8, Table D.9 and Table D.10 summarize the methods used to pass arguments.

Table D.8 Rules on Passing Arguments to Function(NC100)

Type of argument	First argument	Second argument	Third argument	Fourth argument	fifth and following arguments
_Bool type, char	R0L register	R1L register	Stack	Stack	Stack
int (16 bits) short	R0 register	R1 register	R4 register	R5 register	Stack
int (32 bits) float, long	R2R0 register	R3R1 register	R6R4 register	R7R5 register	Stack
pointer	A0 register	A2 register	R6R4 register	R7R5 register	Stack
long long double long double	A1A0 register	A3A2 register	Stack	Stack	Stack

Table D.9 Rules on Passing Arguments to Function(NC308)

Type of argument	First argument	Second argument	Third and following arguments
_Bool char	R0L register	Stack	Stack
int near pointer	R0 register	Stack	Stack

Table D.10 Rules on Passing Arguments to Function(NC30)

Type of argument	First argument	Second argument	Third and following arguments
_Bool char	R1L register	R2 register	Stack
int near pointer	R1 register	Stack	Stack

D.3.3 Rules for Converting Functions into Assembly Language Symbols

The function names in which functions are defined in a C language source file are used as the start labels of functions in an assembler source file.
The beginning label of a function in an assembler source file consists of the function name in the C language source file that is prefixed by an underbar (_) or dollar mark ($), or the function name itself. The appended strings and the conditions under which strings are appended are shown in Table D.11.

Table D.11 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[3]

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.

[3] However, function names are not output for the functions that are specified by #pragma INTCALL.

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

int         func_proto(int i, int j, int k)                        [2]
{
            return k + j + i
}

int         func_no_proto( i, j, k)                                [3]
int         i;
int         j;
int         k;
{
            return k + j + i
}

void        main(void)                                             [4]
{
            int          sum;

            sum = func_proto(1,2,3);                      ← [5]
            sum = func_no_proto(1,2,3);                   ← [6]
}
```

[1] This is the prototype declaration of function func_proto.
[2] This is the body of function func_proto. (Prototype declaration is entered, so this is a new format.)
[3] This is the body of function func_no_proto. (This is a description in K&R format, that is, an old format.)
[4]This is the body of function main.
[5] This calls function func_proto.
[6] This calls function func_no_proto.

Figure D.10 Sample Program for Calling a Function (sample.c)

The compile result of the above sample program is shown in the next page. Figure D.11 shows the compile result of program part [2] that defines function func_proto. Figure D.12 shows the compile result of program part [3] that defines function func_no_proto. Figure D.13 shows the compile result of program part [4] that calls function func_proto and function func_no_proto.

```
;## #        FUNCTION func_proto
;## #   REGISTER ARG      (          i)     size     4,     REGISTER R2R0          ← [9]
;## #   REGISTER ARG      (          j)     size     4,     REGISTER R3R1          ← [8]
;## #   REGISTER ARG      (          k)     size     4,     REGISTER R6R4          ← [7]
;## #        ARG Size(0)              Auto Size(0)              Context Size(4)

             .SECTION  program,CODE,ALIGN
             ._file      'test.c'
             .align
             ._line      4
;## # C_SRC :             {
             .glb        $func_proto
$func_proto:                                          ← [10]
             ._line      5
;## # C_SRC :                          return k + j + i;
             add.l       R3R1,R6R4              ;   j
             add.l       R6R4,R2R0              rts
E1:

[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 compilation result (1) of the sample program (sample.c) in Figure D.10, the first, second, and third arguments are passed via registers because the function func_proto has its prototype declared.
Furthermore, since the arguments to the function are passed via registers, the symbol name for the beginning address of the function is taken after "func_proto" written in the C language source file by prefixing it with the dollar mark ($), namely "$func_proto."

```
;## #        FUNCTION func_no_proto
;## #  FRAME    ARG (        i)   size    4,      offset 8      [11]
;## #  FRAME    ARG (        j)   size    4,      offset 12
;## #  FRAME    ARG (        k)   size    4,      offset 16
;## #        ARG Size(12)          Auto Size(0)           Context Size(8)

             .align
             ._line       12
;## # C_SRC :             {
             .glb         _func_no_proto
_func_no_proto:                                       ← [12]
             enter        #00H
             ._line       13
;## # C_SRC :                     return k + j + i
             mov.l        16[FB],R2R0           ;   k
             add.l        12[FB],R2R0           ;   j
             add.l        8[FB],R2R0;   i
             exitd
E2:

[11] This passes all arguments via a stack.
[12] This is the start address of function func_no_proto.
```

Figure D.12 Compile Result of Sample Program (sample.c) (2/3)

In the compile result (2/3) 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   4,    offset -4
;###        ARG Size(4)         Auto Size(4)             Context Size(8)

            .align
            ._line    17
;### C_SRC :          {
            .glb      _main
_main:
            enter     #04H
            ._line    20
;### C_SRC :                    sum = func_proto(1,2,3);
            mov.l     #00000003H,R6R4                                  [13]
            mov.l     #00000002H,R3R1
            mov.l     #00000001H,R2R0
            jsr       $func_proto
            mov.l     R2R0,-4[FB]         ;_ sum
            ._line    21
;### C_SRC :                    sum = func_no_proto(1,2,3);
            push.l    #00000003H                                       [14]
            push.l    #00000002H
            push.l    #00000001H
            jsr       _func_no_proto
            add.l     #0cH,SP
            mov.l     R2R0,-4[FB]         ;_ sum
            ._line    22
;### C_SRC :          }
            exitd
E3:

            .END
```

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

Figure D.13, part [13] calls func_proto and part [14] calls func_no_proto.

D.3.4 Interface between Functions

Figure D.17 and Figure D.18 show the process for building and freeing the stack frame in the program shown in Figure D.14. Shown in Figure D.15 and Figure D.16 are the assembly language programs derived by compiling the program in Figure D.14.

```
iint func( int, int, int);

void        main(void)
{
            int ans;
            int i = 0x1111;         ← Argument to func
            int j = 0x2222;         ← Argument to func
            int k = 0x3333;         ← Argument to func
            ans = func( i, j ,k);
}

int func( int x, int y, int z )
{
            int sum;
            int s = 0x4444;
            int t = 0x5555;
            int u = 0x6666;
            sum = s + t + u + x + y + z ;
            return sum;             ← Return value to main
}
```

Figure D.14 Example of C Language Sample Program

```
;###        FUNCTION main
;###        FRAME      AUTO       (      ans)  size   4,    offset -4
;###        ARG Size(4)           Auto Size(4)             Context Size(8)

            .SECTION  program,CODE,ALIGN
            ._file     'interface.c'
            .align
            ._line     4
;### C_SRC :           {
            .glb       _main
_main:                                              ← [1]
            enter      #04H                         ← [2]
            ._line     6
;### C_SRC :           int i = 0x1111;
            mov.l      #00001111H,R2R0      ;  i
            ._line     7
;### C_SRC :           int j = 0x2222;
            mov.l      #00002222H,R3R1      ;  j
            ._line     8
;### C_SRC :           int k = 0x3333;
            mov.l      #00003333H,R6R4      ;  k
            ._line     9
;### C_SRC :           ans = func( i, j ,k);
            jsr        $func                        ← [3]
            mov.l      R2R0,-4[FB]          ;  ans  ← [7]
            ._line     10
;### C_SRC :           }
            exitd
E1:
```

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

```
;## #        FUNCTION func
;## #        FRAME    AUTO      (         u)  size   4,     offset -4
;## #   REGISTER ARG      (         x)    size     4,    REGISTER R2R0
;## #   REGISTER ARG      (         y)    size     4,    REGISTER R3R1
;## #   REGISTER ARG      (         z)    size     4,    REGISTER R6R4
;## #        ARG Size(0)          Auto Size(4)            Context Size(8)

            .align
            ._line      13
;## # C_SRC :           {
            .glb        $func
$func:
            enter       #04H                            ← [4]
            ._line      15
;## # C_SRC :           int s = 0x4444;
            mov.l       #00004444H,R7R5     ;   s
            ._line      16
;## # C_SRC :           int t = 0x5555;
            mov.l       #00005555H,A0       ;   t
            ._line      17
;## # C_SRC :           int u = 0x6666;
            mov.l       #00006666H,-4[FB]   ;   u
            ._line      18
;## # C_SRC :           sum = s + t + u + x + y + z ;
            add.l       A0,R7R5   ;   t
            add.l       -4[FB],R7R5         ;   u
            add.l       R7R5,R2R0
            add.l       R2R0,R3R1
            mov.l       R6R4,R2R0           ;   z   z
            add.l       R3R1,R2R0           ;   sum     ← [5]
            ._line      19
;## # C_SRC :           return sum;
            exitd                                       ←[6]
E2:
```

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

Figure D.17 and Figure D.18 show the stack and register behaviors during the processes [1], [2], and [3] in Figure D.15 (i.e., process at entry to the function main and process to call the function func) and during the processes [4], [5], [6], and [7] (i.e., process to build the stack frame used in the function func and process to return from the function func to the function main), respectively.

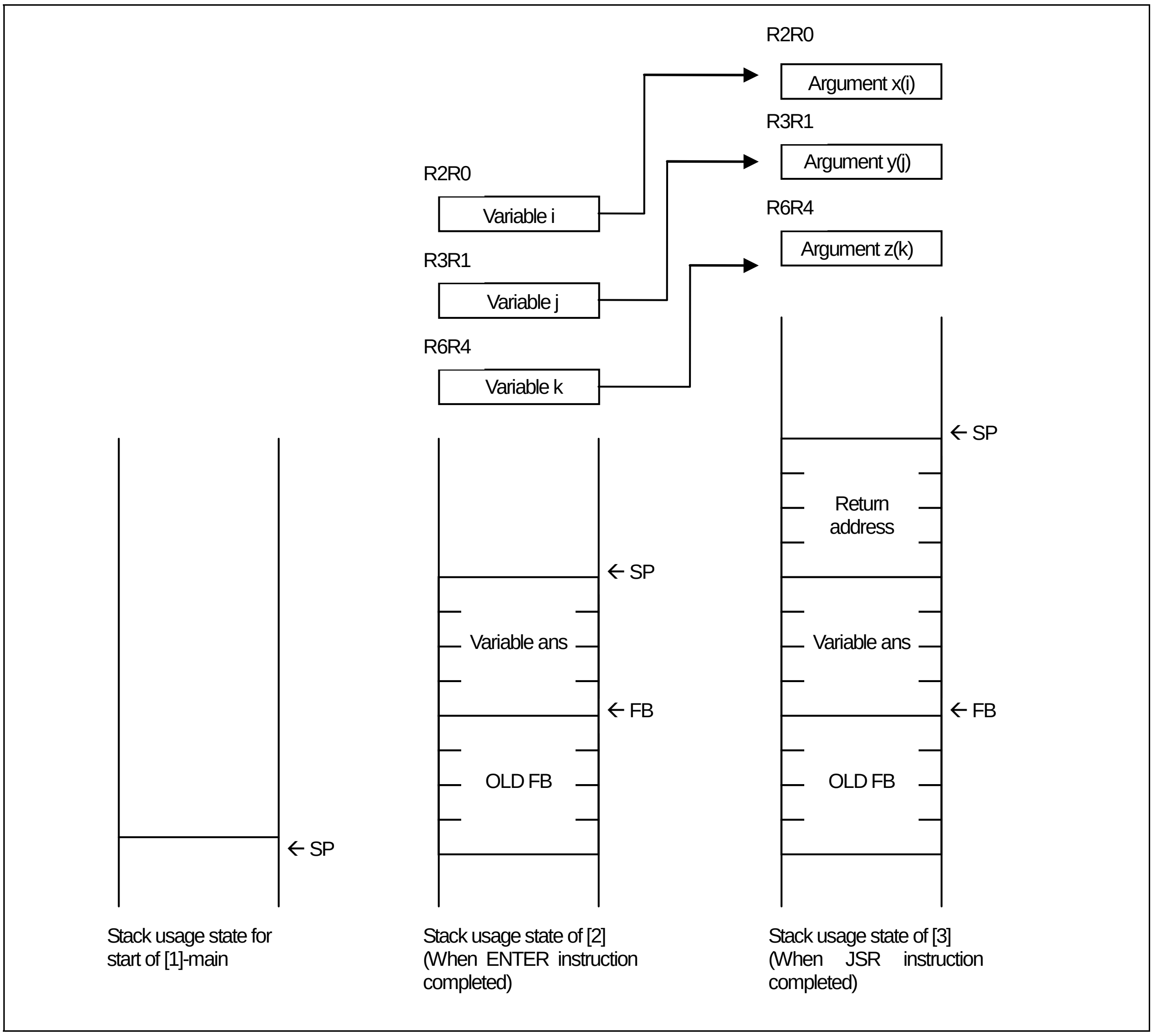


Figure D.17 Process at entry to the function and process to call the function func

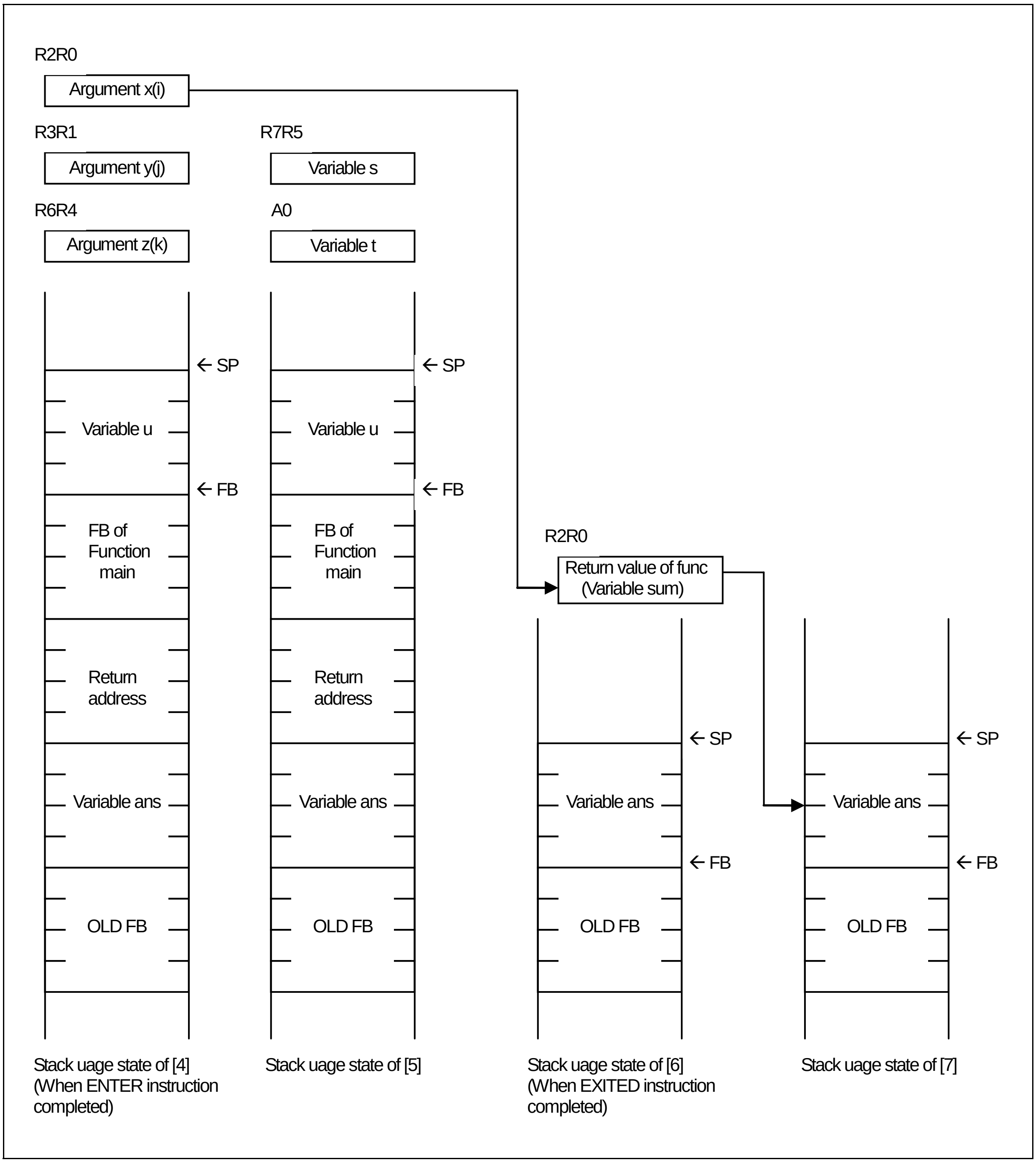


Figure D.18 Process to build the stack frame used in the function func and process to return from the function func to the function main

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.19, if the areas where variables of storage class auto are valid do not overlap each other, the system allocates only one area which is then shared between multiple variables.

```
void        func(void)
{
            int         i, j, k;
            for( i=0 ; i<=0 ; i++ ){
                        process                             scope of i
            }
                            :
                        (abbreviated)
                            :
            for( j=0xFF ; j<=0 ; j-- ){
                        process                             scope of j
                        }
                            :
                        (abbreviated)
                            :
            for( k=0 ; k<=0 ; k++ ){
                        process                             scope of k
            }
}
```

Figure D.19 Example of C Program

In 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.20 shows an assembly language source file generated by compiling the program in Figure D.19.

```
;### FUNCTION        func
;###        FRAME AUTO   (        k)    size 4,    offset -4          ← [1]
;###        FRAME AUTO   (        j)    size 4,    offset -4          ← [2]
;###        FRAME AUTO   (       i)     size 4,    offset -4          ← [3]
            .section program
            ._file      'auto.c'
            ._line      3
            .glb        _func
_func:
            enter       #04H
                    :
            (remainder omitted)

* As shown by [1],[2], and [3],the three auto variables share the FB offset -2 area.
```

Figure D.20 Example of Assembly Language Source Program

D.5 Rules of Escaping of the Register

The rules of Escaping of the register when call C function as follows:

(1) The rules of Escaping of the register when call C function as follows:
- Register which use in called C function

(2) Register which should escaping in the entrance procedure of the called function.
- None

Appendix E Standard Library

E.1 Standard Header Files

When using the NC100 standard library, you must include the header file that defines that function. This appendix details the functions and specifications of the standard NC100 header files.

E.1.1 Contents of Standard Header Files

NC100 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	(1) Defines the FILE structure. (2) Defines a stream name. (3) Declares the prototype of input/output functions.
stdlib.h	Declares the prototypes of memory management and terminate functions.
string.h	Declares the prototypes of character string and memory handling functions.
time.h	Declares the functions necessary to indicate the current calendar time and defines the type.

E.1.2 Standard Header Files Reference

Following are detailed descriptions of the standard header files supplied with NC100. The header files are presented in alphabetical order.
The NC100 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 NC100, 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 or 127(When a compile option "-fSC(-fsigned_char)" is specified)
CHAR_MIN	Minimum value that a char-type variable can take on as value	0 or -128(When a compile option "-fSC(-fsigned_char)" is specified)
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 or 2147483647(When a compile option "-fI16(-fint_16)" is specified)
INT_MIN	Minimum value that a int-type variable can take on as value	32768 or 2147483648(When a compile option "-fI16(-fint_16)" is specified)
SHRT_MAX	Maximum value that a short int-type variable can take on as value	32767
SHRT_MIN	Minimum value that a short int-type variable can take on as value	-32768
LONG_MAX	Maximum value that a long-type variable can take on as value	2147483647
LONG_MIN	Minimum value that a long-type variable can take on as value	-2147483648
LLONG_MAX	Maximum value that a signed long long-type variable can take on as value	9223372036854775807
LLONG_MIN	Minimum value that a signed long longtype variable can take on as value	-9223372036854775808
UCHAR_MAX	Maximum value that an unsigned char-type variable can take on as value	255
UINT_MAX	Maximum value that an unsigned int-type variable can take on as value	65535 or 4294967295 (When a compile option "-fI16(-fint_16)" is specified)
USHRT_MAX	Maximum value that an unsigned short int-type variable can take on as value	65535
ULONG_MAX	Maximum value that an unsigned long int-type variable can take on as value	4294967295
ULLONG_MAX	Maximum value that an unsigned long long inttype variable can take on as value	18446744073709551615

locale.h

Function: Defines/declares macros and functions that manipulate program localization.The following lists locale functions.

Function	Contents
localeconv	Initializes struct lconv.
setlocale	Sets and searches the locale information of a program.

math.h

Function: Declares prototype of mathematical function.The following lists mathematical functions.

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

setjmp.h

Function: Defines the structures used in branch functions.

Function	Contents
longjmp	Performs a global jump.
setjmp	Sets a stack environment for a global jump.

signal.h

Function: Defines/declares necessary for processing asynchronous interrupts.

stdarg.h

Function: Defines/declares the functions which have a variable number of real arguments.

stddef.h

Function: Defines the macro names which are shared among standard include files.

stdio.h

Function: Defines the FILE structure, stream name, and declares I/O function prototypes. Prototype declarations are made for the following functions.

Type	Function	Function
Initialize	init	Initializes R32C/100 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.
Deter-mination	ferror	Checks input/output errors.
	feof	Checks EOF (End of File).

stdlib.h

Function: Declares the prototypes of memory management and terminate functions.

Function	Contents
abort	Terminates the execution of the program.
abs	Calculates the absolute value of an integer.
atof	Converts a character string into a double-type floating-point number.
atoi	Converts a character string into an int-type integer.
atol	Converts a character string into a long-type integer.
bsearch	Performs binary search in an array.
calloc	Allocates a memory area and initializes it to zero (0).
div	Divides an int-type integer and calculates the remainder.
free	Frees the allocated memory area.
labs	Calculates the absolute value of a long-type integer.
ldiv	Divides a long-type integer and calculates the remainder.
malloc	Allocates a memory area.
mblen	Calculates the length of a multibyte character string.
mbstowcs	Converts a multibyte character string into a wide character string.
mbtowc	Converts a multibyte character into a wide character.
qsort	Sorts elements in an array.
realloc	Changes the size of an allocated memory area.
strtod	Converts a character string into a double-type integer.
strtol	Converts a character string into a long-type integer.
strtoul	Converts a character string into an unsigned long-type integer.
wcstombs	Converts a wide character string into a multibyte character string.
wctomb	Converts a wide character into a multibyte character.

string.h

Function: Declares the prototypes of string handling functions and memory handling functions.

Type	Type	Contents
Copy	strcpy	Copies a character string.
	strncpy	Copies a character string ('n' characters).
Concatenate	strcat	Concatenates character strings.
	strncat	Concatenates character strings ('n' characters).
Compare	strcmp	Compares character strings.
	strcoll	Compares character strings (using locale information).
	stricmp	Compares character strings. (All alphabets are handled as upper-case letters.)
	strncmp	Compares character strings ('n' characters).
	strnicmp	Compares character strings ('n' characters). (All alphabets are handled as upper-case letters.)
Search	strchr	Searches the specified character beginning with the top of the character string.
	strcspn	Calculates the length (number) of unspecified characters that are not found in the other character string.
	strpbrk	Searches the specified character in a character string from the other character string.
	strrchr	Searches the specified character from the end of a character string.
	strspn	Calculates the length (number) of specified characters that are found in the other character string.
	strstr	Searches the specified character from a character string.
	strtok	Divides some character string from a character string into tokens.
Length	strlen	Calculates the number of characters in a character string.
Convert	strerror	Converts an error number into a character string.
	strxfrm	Converts a character string (using locale information).
Initialize	bzero	Initializes a memory area (by clearing it to zero).
Copy	bcopy	Copies characters from a memory area to another.
	memcpy	Copies characters ('n' bytes) from a memory area to another.
	memset	Set a memory area by filling with characters.
Compare	memcmp	Compares memory areas ('n' bytes).
	memicmp	Compares memory areas (with alphabets handled as uppercase letters).
Search	memchr	Searches a character from a memory area.

time.h

Function: Declares the functions necessary to indicate the current calendar time and defines the type.

E.2 Standard Function Reference

Describes the features and detailed specifications of the standard function library of the compiler.

E.2.1 Overview of Standard Library

NC100 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
* 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.	O
	strncpy	Copies a character string ('n' characters).	O
Concatenate	strcat	Concatenates character strings.	O
	strncat	Concatenates character strings ('n' characters).	O
Compare	strcmp	Compares character strings.	O
	strcoll	Compares character strings (using locale information).	O
	stricmp	Compares character strings. (All alphabets are handled as upper-case letters.)	O
	strncmp	Compares character strings ('n' characters).	O
	strnicmp	Compares character strings ('n' characters). (All alphabets are handled as upper-case letters.)	O
Search	strchr	Searches the specified character beginning with the top of the character string.	O
	strcspn	Calculates the length (number) of unspecified characters that are not found in the other character string.	O
	strpbrk	Searches the specified character in a character string from the other character string.	O
	strrchr	Searches the specified character from the end of a character string.	O
	strspn	Calculates the length (number) of specified characters that are found in the other character string.	O
	strstr	Searches the specified character from a character string.	O
	strtok	Divides some character string from a character string into tokens.	X
Length	strlen	Calculates the number of characters in a character string.	O
Convert	strerror	Converts an error number into a character string.	X
	strxfrm	Converts a character string (using locale information).	O

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.	O
isalpha	Checks whether the character is an alphabet.	O
iscntrl	Checks whether the character is a control character.	O
isdigit	Checks whether the character is a numeral.	O
isgraph	Checks whether the character is printable (except a blank).	O
islower	Checks whether the character is a lower-case letter.	O
isprint	Checks whether the character is printable (including a blank).	O
ispunct	Checks whether the character is a punctuation character.	O
isspace	Checks whether the character is a blank, tab, or new line.	O
isupper	Checks whether the character is an upper-case letter.	O
isxdigit	Checks whether the character is a hexadecimal character.	O
tolower	Converts the character from an upper-case to a lowercase.	O
toupper	Converts the character from a lower-case to an uppercase.	O

c Input/Output Functions

The following lists Input/Output functions.

Table E.4 Input/Output Functions

Type	Function	Contents	Reentrant
Initialize	init	Initializes R32C series's input/outputs.	X
	clearerror	Initializes (clears) error status specifiers.	X
Initialize	fgetc	Inputs one character from the stream.	X
	getc	Inputs one character from the stream.	X
	getchar	Inputs one character from stdin.	X
	fgets	Inputs one line from the stream.	X
	gets	Inputs one line from stdin.	X
	fread	Inputs the specified items of data from the stream.	X
	scanf	Inputs characters with format from stdin.	X
	fscanf	Inputs characters with format from the stream.	X
	sscanf	Inputs data with format from a character string.	X
Output	fputc	Outputs one character to the stream.	X
	putc	Outputs one character to the stream.	X
	putchar	Outputs one character to stdout.	X
	fputs	Outputs one line to the stream.	X
	puts	Outputs one line to stdout.	X
	fwrite	Outputs the specified items of data to the stream.	X
	perror	Outputs an error message to stdout.	X
	printf	Outputs characters with format to stdout.	X
	fflush	Flushes the stream of an output buffer.	X
	fprintf	Outputs characters with format to the stream.	X
	sprintf	Writes text with format to a character string.	X
	vfprintf	Output to a stream with format.	X
	vprintf	Output to stdout with format.	X
	vsprintf	Output to a buffer with format.	X
Return	ungetc	Sends one character back to the input stream.	X
Determination	ferror	Checks input/output errors.	X
	feof	Checks EOF (End of File).	X

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).	X
free	Frees the allocated memory area.	X
malloc	Allocates a memory area.	X
realloc	Changes the size of an allocated memory area.	X

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).	O
Copy	bcopy	Copies characters from a memory area to another.	O
	memcpy	Copies characters ('n' bytes) from a memory area to another.	O
	memset	Set a memory area by filling with characters.	O
Compare	memcmp	Compares memory areas ('n' bytes).	O
	memicmp	Compares memory areas (with alphabets handled as upper-case letters).	O
Move	memmove	Moves the area of a character string.	O
Search	memchr	Searches a character from a memory area.	O

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.	O
longjmp	Performs a global jump.	O
setjmp	Sets a stack environment for a global jump.	O

g Mathematical Functions

The following lists mathematical functions.

Table E.8 Mathematical Functions

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

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

i Character String Value Convert Functions

The following lists character string value convert functions.

Table E.10 Character String Value Convert Functions

Function	Contents	Reentrant
atof	Converts a character string into a double-type floatingpoint number.	O
atoi	Converts a character string into an int	O
atol	Converts a character string into a long	O
strtod	Converts a character string into a double	O
strtol	Converts a character string into a long	O
strtou	Converts a character string into an unsigned long-type integer.	O

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.	O
mbstowcs	Converts a multibyte character string into a wide character string.	O
mbtowc	Converts a multibyte character into a wide character.	O
wcstombs	Converts a wide character string into a multibyte character string.	O
wctomb	Converts a wide character into a multibyte character.	O

k Localization Functions

The following lists localization functions.

Table E.12 Localization Functions

Function	Contents	Reentrant
localeconv	Initializes struct lconv.	O
setlocale	Sets and searches the locale information of a program.	O

E.2.3 Standard Function Reference

The following describes the detailed specifications of the standard functions provided in NC100. The functions are listed in alphabetical order.
Note that the standard header file (extension .h) shown under "Format" must be included when that function is used.

A

abort

Execution Control Functions

Function:	Terminates the execution of the program abnormally.
Format:	`#include<stdlib.h>`
	`void abort(void);`
Method:	function
Variable:	No argument used.
ReturnValue:	No value is returned.
Description:	Terminates the execution of the program abnormally.
Note:	Actually, the program loops in the abort function.

abs

Integer Arithmetic Functions

Function:	Calculates the absolute value of an integer.	
Format:	`#include<stdlib.h>`	
	`int abs(n);`	
Method:	function	
Variable:	int n;	Integer
ReturnValue:	Returns the absolute value of integer n (distance from 0).	

acos

Mathematical Functions

Function: Calculates arc cosine.

Format:
```
#include<math.h>
double acos(x);
```

Method: function

Variable: double x; arbitrary real number

ReturnValue:
- Assumes an error and returns 0 if the value of given real number x is outside therange of -1.0 to 1.0.
- Otherwise, returns a value in the range from 0 to π 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 -π/2 to π/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 $-\pi$ to π radian.

atof

Character String Value Convert Functions

Function: Converts a character string into a double-type floating- point number.

Format:
```
#include <stdlib.h>

double atof(s);
```

Method: function

Variable: const char _far *s; Pointer to the converted character string

ReturnValue: Returns the value derived by converting a character string into a double-precision floating-point number.

atoi

Character String Convert Functions

Function: Converts a character string into an int-type integer.

Format:
```
#include <stdlib.h>

int atoi(s);
```

Method: function

Variable: const char _far *s; Pointer to the converted character string

ReturnValue: Returns the value derived by converting a character string into an int-type integer.

atol

Character String Convert Functions

Function:	Converts a character string into a long-type integer.
Format:	`#include <stdlib.h>`
	`long atol(s);`
Method:	function
Variable:	const char _far *s; Pointer to the converted character string
ReturnValue:	Returns the value derived by converting a character string into a long-type integer.

B

bcopy

Memory Handling Functions

Function: Copies characters from a memory area to another.

Format:

```
#include <string.h>

void bcopy(src, dtop, size);
```

Method: function

Variable:

char _far *src;	Start address of the memory area to be copied from
char _far *dtop;	Start address of the memory area to be copied to
unsigned long size;	Number of bytes to be copied

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

bsearch

Integer Arithmetic Functions

Function: Performs binary search in an array.

Format:

```
#include <stdlib.h>

void _far *bsearch( key, base, nelem, size, cmp );
```

Method: function

Variable:

const void _far *key;	Search key
const void _far *base;	Start address of array
size_t nelem;	Element number
size_t size;	Element size
int cmp();	Compare function

ReturnValue:

- Returns a pointer to an array element that equals the search key.
- Returns a NULL pointer if no elements matched.

Note: The specified item is searched from the array after it has been sorted in ascending order.

bzero

Memory Handling Functions

Function:	Initializes a memory area (by clearing it to zero).	
Format:	`#include <string.h>`	
	`void bzero(top, size);`	
Method:	function	
Variable:	char _far *top;	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.	

C

calloc

Memory Management Functions

Function: Allocates a memory area and initializes it to zero (0).

Format:
```
#include <stdlib.h>

void _far * calloc(n, size);
```

Method: function

Variable:

size_t n;	Number of elements
size_t size;	Value indicating the element size in bytes

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

Description:

- After allocating the specified memory, it is cleared to zero.
- The size of the memory area is the product of the two parameters.

Rule: The rules for securing memory are the same as for malloc.

ceil

Mathematical Functions

Function: Calculates an integer carry value.

Format:
```
#include <math.h>

double ceil(x);
```

Method: function

Argument:

double x;	arbitrary real number

ReturnValue: Returns the minimum integer value from among integers larger than given real number x.

clearerr

Input/Output Functions

Function: Initializes (clears) error status specifiers.

Format:
```
#include <stdio.h>
void clearerr(stream);
```

Method: function

Argument: FILE _far *stream; Pointer of stream

ReturnValue: No value is returned.

Description: Resets the error designator and end of file designator to their normal values.

cos

Mathematical Functions

Function: Calculates cosine.

Format:
```
#include <math.h>
double cos(x);
```

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the cosine of given real number x handled in units of radian.

cosh

Mathematical Functions

Function: Calculates hyperbolic cosine.

Format:
```
#include <math.h>
double cosh(x);
```

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the hyperbolic cosine of given real number x.

D

div

Integer Arithmetic Functions

Function: Divides an int-type integer and calculates the remainder.

Format:
```
#include <stdlib.h>

div_t div(number, denom);
```

Method: function

Argument:

int number;	Dividend
int denom;	Divisor

ReturnValue: Returns the quotient derived by dividing "number" by "denom" and the remainder of the division.

Description:
- Returns the quotient derived by dividing "number" by "denom" and the remainder of the division in structure div_t.
- div_t is defined in stdlib.h. This structure consists of members int quot and int rem.

E

exp

Mathematical Functions

Function: Calculates exponential function.

Format:
```
#include <math.h>

double exp(x);
```

Method: function

Argument:

double x;	arbitrary real number

ReturnValue: Returns the calculation result of an exponential function of given real number x.

F

fabs

Mathematical Functions

Function: Calculates the absolute value of a double-precision floating-point number.

Format:
```
#include <math.h>
double fabs(x);
```

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the absolute value of a double-precision floating-point number.

feof

Input/Output Functions

Function: Checks EOF (End of File).

Format:
```
#include <stdio.h>
int feof(stream);
```

Method: macro

Argument: FILE _far *stream; Pointer of stream

ReturnValue:
- Returns "true" (other than 0) if the stream is EOF.
- Otherwise, returns NULL (0).

Description:
- Determines if the stream has been read to the EOF.
- Interprets code 0x1A as the end code and ignores any subsequent data.

ferror

Input/Output Functions

Function: Checks input/output errors.

Format:
```
#include <stdio.h>

int ferror(stream);
```

Method: macro

Argument: FILE _far *stream; Pointer of stream

ReturnValue:
- Returns "true" (other than 0) if the stream is in error.
- Otherwise, returns NULL (0).

Description:
- Determines errors in the stream.
- Interprets code 0x1A as the end code and ignores any subsequent data.

fflush

Input/Output Functions

Function: Flushes the stream of an output buffer.

Format:
```
#include <stdio.h>

int fflush(stream);
```

Method: function

Argument: FILE _far *stream; Pointer of stream

ReturnValue: Always returns 0.

fgetc

Input/Output Functions

Function: Reads one character from the stream.

Format:
```
#include <stdio.h>

int fgetc(stream);
```

Method: function

Argument: FILE _far *stream; Pointer of stream

ReturnValue:
- Returns the one input character.
- Returns EOF if an error or the end of the stream is encountered.

Description:
- Reads one character from the stream.
- Interprets code 0x1A as the end code and ignores any subsequent data.

fgets

Input/Output Functions

Function: Reads one line from the stream.

Format:
```
#include <stdio.h>

char _far * fgets(buffer, n, stream);
```

Method: function

Argument:

char _far *buffer;	Pointer of the location to be stored in
int n;	Maximum number of characters
FILE _far *stream;	Pointer of stream

ReturnValue:
- Returns the pointer of the location to be stored (the same pointer as given by the argument) if normally input.
- Returns the NULL pointer if an error or the end of the stream is encountered.

Description:
- Reads character string from the specified stream and stores it in the buffer
- Input ends at the input of any of the following:
  (1) new line character ('¥n')
  (2) n-1 characters
  (3) end of stream
- A null character ('¥0') is appended to the end of the input character string.
- The new line character ('¥n') is stored as-is.
- Interprets code 0x1A as the end code and ignores any subsequent data.

floor

Mathematical Functions

Function:	Calculates an integer borrow value.	
Format:	`#include <math.h>` `double floor(x);`	
Method:	function	
Argument:	double x;	arbitrary real number
ReturnValue:	The real value is truncated to form an integer, which is returned as a double type.	

fmod

Mathematical Functions

Function:	Calculates the remainder.	
Format:	`#include <math.h>` `double fmod(x, y);`	
Method:	function	
Argument:	double x; double y;	dividend divisor
ReturnValue:	Returns a remainder that derives when dividend x is divided by divisor y.	

fprintf

Input/Output Functions

Function: Outputs characters with format to the stream.

Format:

```
#include <stdio.h>
int fprintf(stream, format, argument...);
```

Method: function

Argument:

FILE _far *stream;	Pointer of stream
const char _far *format;	Pointer of the format specifying character string

ReturnValue:

- Returns the number of characters output.
- Returns EOF if a hardware error occurs.

Description:

- Argument is converted to a character string according to format and output to the stream.
- Format is specified in the same way as in printf.

fputc

Input/Output Functions

Function: Outputs one character to the stream.

Format:
```
#include <stdio.h>

int fputc(c, stream);
```

Method: function

Argument:

int c;	Character to be output
FILE _far *stream;	Pointer of the stream

ReturnValue:
- Returns the output character if output normally.
- Returns EOF if an error occurs.

Description: Outputs one character to the stream.

fputs

Input/Output Functions

Function: Outputs one line to the stream.

Format:
```
#include <stdio.h>

int fputs (str, stream);
```

Method: function

Argument:

const char _far *str;	Pointer of the character string to be output
FILE _far *stream;	Pointer of the stream

ReturnValue:
- Returns 0 if output normally.
- Returns any value other than 0 (EOF) if an error occurs.

Description: Outputs one line to the stream.

fread

Input/Output Functions

Function:	Reads fixed-length data from the stream
Format:	`#include <stdio.h>` `size_t fread(buffer, size, count, stream);`
Method:	function

Argument:

void _far *buffer;	Pointer of the location to be stored in
size_t size;	Number of bytes in one data item
size_t count;	Maximum number of data items
FILE _far *stream;	Pointer of stream

ReturnValue: Returns the number of data items input.

Description:

- Reads data of the size specified in size from the stream and stores it in the buffer. This is repeated by the number of times specified in count.
- If the end of the stream is encountered before the data specified in count has been input, this function returns the number of data items read up to the end of the stream.
- Interprets code 0x1A as the end code and ignores any subsequent data.

free

Memory Management Function

Function:	Frees the allocated memory area.
Format:	`#include <stdlib.h>` `void free(cp);`
Method:	function
Argument:	void _far *cp; — Pointer to the memory area to be freed
ReturnValue:	No value is returned.

Description:

- Frees memory areas previously allocated with malloc or calloc.
- No processing is performed if you specify NULL in the parameter.

frexp

Mathematical Functions

Function: Divides floating-point number into mantissa and exponent parts.

Format:
```
#include <math.h>

double frexp(x, prexp);
```

Method: function

Argument:

double x;	float-point number
int _far *prexp;	Pointer to an area for storing a 2-based exponent

ReturnValue Returns the floating-point number x mantissa part.

fscanf

Input/Output Function

Function: Reads characters with format from the stream.

Format:
```
#include <stdio.h>

int fscanf( stream, format, argument...);
```

Method: function

Argument:

FILE _far *stream;	Pointer of stream
const char _far *format;	Pointer of the input character string

ReturnValue:

- Returns the number of data entries stored in each argument.
- Returns EOF if EOF is input from the stream as data.

Description:

- Converts the characters input from the stream as specified in format and stores them in the variables shown in the arguments.
- Argument must be a pointer to the respective variable.
- Interprets code 0x1A as the end code and ignores any subsequent data.
- Format is specified in the same way as in scanf.

fwrite

Input/Output Functions

Function: Outputs the specified items of data to the stream.

Format:

```
#include <stdio.h>

size_t fwrite(buffer, size, count, stream);
```

Method: function

Argument:

const void _far *buffer;	Pointer of the output data
size_t size;	Number of bytes in one data item
size_t count;	Maximum number of data items
FILE _far *stream;	Pointer of the stream

ReturnValue: Returns the number of data items output

Description:

- Outputs data with the size specified in size to the stream. Data is output by the number of times specified in count.
- If an error occurs before the amount of data specified in count has been input, this function returns the number of data items output to that point.

G

getc

Input/Output Functions

Function: Reads one character from the stream.

Format:
```
#include <stdio.h>

int getc(stream);
```

Method: macro

Argument: FILE _far *stream; Pointer of stream

ReturnValue:
- Returns the one input character.
- Returns EOF if an error or the end of the stream is encountered.

Description:
- Reads one character from the stream.
- Interprets code 0x1A as the end code and ignores any subsequent data.

getchar

Input/Output Functions

Function: Reads one character from stdin.

Format:
```
#include <stdio.h>

int getchar(void);
```

Method: macro

Argument: No argument used.

ReturnValue:
- Returns the one input character.
- Returns EOF if an error or the end of the file is encountered.

Description:
- Reads one character from stream (stdin).
- Interprets code 0x1A as the end code and ignores any subsequent data.

gets

Input/Output Functions

Function: Reads one line from stdin.

Format:
```
#include <stdio.h>
char _far * gets(buffer);
```

Method: function

Argument: char _far *buffer; Pointer of the location to be stored in

ReturnValue:
- Returns the pointer of the location to be stored (the same pointer as given by the argument) if normally input.
- Returns the NULL pointer if an error or the end of the file is encountered.

Description:
- Reads character string from stdin and stores it in the buffer.
- The new line character ('¥n') at the end of the line is replaced with the null character ('¥0').
- Interprets code 0x1A as the end code and ignores any subsequent data.

I

init

Input/Output Functions

Function: Initializes the stream.

Format:
```
#include <stdio.h>
void init(void);
```

Method: function

Argument: No argument used.

ReturnValue: No value is returned.

Description:
- Initializes the stream. Also calls speed and init_prn in the function to make the initial settings of the UART and Centronics output device.
- init is normally used by calling it from the startup program.

isalnum

Character Handling Functions

Function: Checks whether the character is an alphabet or numeral (A - Z,a - z,0 - 9).

Format:
```
#include <ctype.h>
int isalnum(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if an alphabet or numeral.
- Returns 0 if not an alphabet nor numeral.

Description: Determines the type of character in the parameter.

isalpha

Character Handling Functions

Function: Checks whether the character is an alphabet(A - Z,a - z).

Format:
```
#include <ctype.h>
int isalpha(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if an alphabet.
- Returns 0 if not an alphabet.

Description: Determines the type of character in the parameter.

iscntrl

Character Handling Functions

Function: Checks whether the character is a control character(0x00 - 0x1f,0x7f).

Format:
```
#include <ctype.h>
int iscntrl(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a numeral.
- Returns 0 if not a control character.

Description: Determines the type of character in the parameter.

isdigit

Character Handling Functions

Function: Checks whether the character is a numeral(0 - 9).

Format:
```
#include <ctype.h>
int isdigit(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a numeral.
- Returns 0 if not a numeral.

Description: Determines the type of character in the parameter.

isgraph

Character Handling Functions

Function: Checks whether the character is printable (except a blank)(0x21 - 0x7e).

Format:
```
#include <ctype.h>
int isgraph(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if printable.
- Returns 0 if not printable.

Description: Determines the type of character in the parameter.

islower

Character Handling Functions

Function: Checks whether the character is a lower-case letter (a - z).

Format:
```
#include <ctype.h>
int islower(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a lower-case letter.
- Returns 0 if not a lower-case letter.

Description: Determines the type of character in the parameter.

isprint

Character Handling Functions

Function: Checks whether the character is printable (including a blank) (0x20 - 0x7e).

Format:
```
#include <ctype.h>
int isprint(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if printable.
- Returns 0 if not printable.

Description: Determines the type of character in the parameter.

ispunct

Character Handling Functions

Function: Checks whether the character is a punctuation character.

Format:
```
#include <ctype.h>
int ispunct(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a punctuation character.
- Returns 0 if not a punctuation character.

Description: Determines the type of character in the parameter.

isspace

Character Handling Functions

Function: Checks whether the character is a blank, tab, or new line.

Format:
```
#include <ctype.h>
int isspace(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a blank, tab, or new line.
- Returns 0 if not a blank, tab, or new line.

Description: Determines the type of character in the parameter.

isupper

Character Handling Functions

Function: Checks whether the character is an upper-case letter (A - Z).

Format:
```
#include <ctype.h>

int isupper(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if an upper-case letter.
- Returns 0 if not an upper-case letter.

Description: Determines the type of character in the parameter.

isxdigit

Character Handling Functions

Function: Checks whether the character is a hexadecimal character (0 - 9, A - F, a - f).

Format:
```
#include <ctype.h>

int isxdigit(c);
```

Method: macro

Argument: int c; Character to be checked

ReturnValue:
- Returns any value other than 0 if a hexadecimal character.
- Returns 0 if not a hexadecimal character.

Description: Determines the type of character in the parameter.

L

labs

Integer Arithmetic Functions

Function: Calculates the absolute value of a long-type integer.

Format:
```
#include <stdlib.h>
long labs(n);
```

Method: function

Argument: long n; Long integer

ReturnValue: Returns the absolute value of a long-type integer (distance from 0).

ldexp

Localization Functions

Function: Calculates the power of a floating-point number.

Format:
```
#include <math.h>
double ldexp(x,exp);
```

Method: function

Argument:
double x; Float-point number
int exp; Power of number

ReturnValue: Returns x *(exp power of 2).

ldiv

Integer Arithmetic Functions

Function: Divides a long-type integer and calculates the remainder.

Format:
```
#include <stdlib.h>

ldiv_t ldiv(number, denom);
```

Method: function

Argument:

long number;	Dividend
long denom;	Divisor

ReturnValue: Returns the quotient derived by dividing "number" by "denom" and the remainder of the division.

Description:
- Returns the quotient derived by dividing "number" by "denom" and the remainder of the division in the structure ldiv_t.
- ldiv_t is defined in stdlib.h. This structure consists of members long quot and longrem.

localeconv

Localization Functions

Function: Initializes struct lconv.

Format:
```
#include <locale.h>

struct lconv _far *localeconv(void);
```

Method: function

Argument: No argument used.

ReturnValue: Returns a pointer to the initialized struct lconv.

log

Mathematical Functions

Function:	Calculates natural logarithm.
Format:	`#include <math.h>`
	`double log(x);`
Method:	function
Argument:	double x; arbitrary real number
ReturnValue:	Returns the natural logarithm of given real number x.
Description:	This is the reverse function of exp.

log10

Mathematical Functions

Function:	Calculates common logarithm.
Format:	`#include <math.h>`
	`double log10(x);`
Method:	function
Argument:	double x; arbitrary real number
ReturnValue:	Returns the common logarithm of given real number 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 "val" is returned as the result of setjmp. However, if "val" is "0", it is converted to "1".

M

malloc

Memory Management Functions

Function: Allocates a memory area.

Format:
```
#include <stdlib.h>

void _far * malloc(nbytes);
```

Method: function

Argument: size_t nbytes; Size of memory area (in bytes) to be allocated ....

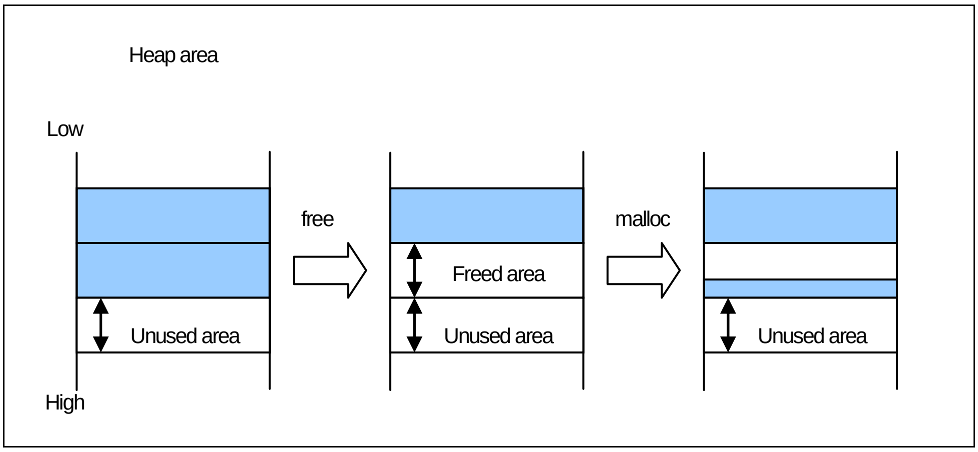
ReturnValue: Returns NULL if a memory area of the specified size could not be allocated.

Description: Dynamically allocates memory areas

Rule: malloc performs the following two checks to secure memory in the appropriate location.

(1) If memory areas have been freed with free

- If the amount of memory to be secured is smaller than that freed, the area is secured from the high address of the contiguously empty area created by free toward the low address.

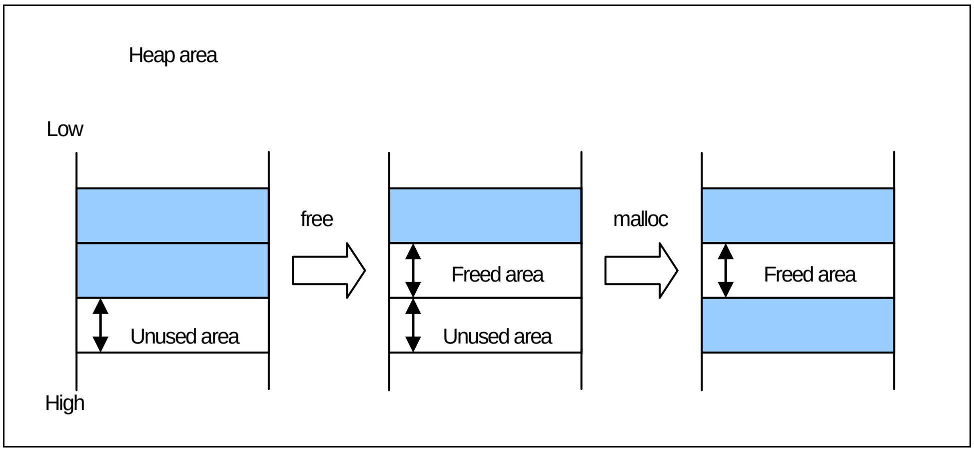


malloc

Memory Management Functions

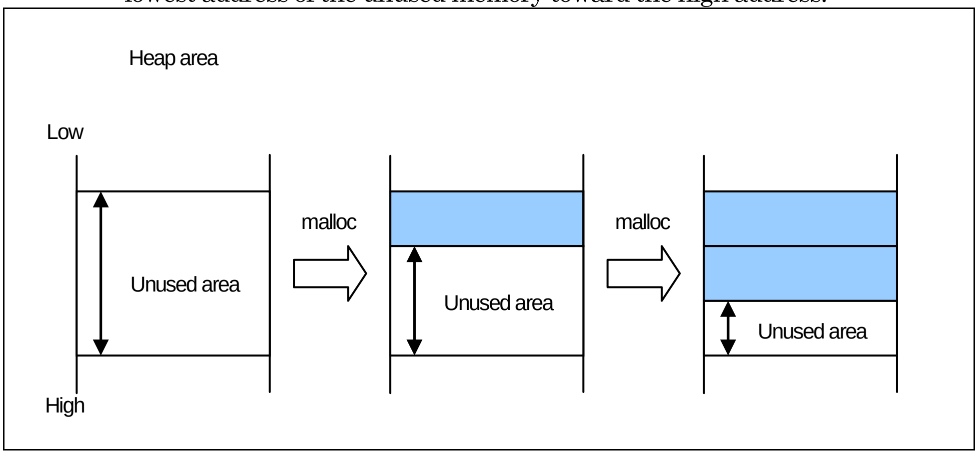
Rule:

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



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

- If there is any unused area that can be secured, the area is secured from the lowest address of the unused memory toward the high address.



- If there is no unused area that can be secured, malloc returns NULL without any memory being secured.

Note: No garbage collection is performed. Therefore, even if there are lots of small unused portions of memory, no memory is secured and malloc returns NULL unless there is an unused portion of memory that is larger than the specified size.

mblen

Multi-byte Character Multi-byte Character String Manipulate Functions

Function: Calculates the length of a multibyte character string.

Format:
```
#include <stdlib.h>

int mblen (s, n);
```

Method: function

Argument:

const char _far *s;	Pointer to a multibyte character string
size_t n;	Number of searched byte

ReturnValue:
- Returns the number of bytes in the character string if 's' configures a correct multibyte character string.
- Returns -1 if 's' does not configure a correct multibyte character string.

Description:
- Returns 0 if 's' indicates a NULL character.

mbstowcs

Multi-byte Character Multi-byte Character String Manipulate Functions

Function: Converts a multibyte character string into a wide character string.

Format:
```
#include <stdlib.h>

size_t mbstowcs(wcs, s, n);
```

Method: function

Argument:

wchar_t _far *wcs;	Pointer to an area for storing conversion wide character string
const char _far *s;	Pointer to a multibyte character string
size_t n;	Number of wide characters stored

ReturnValue:
- Returns the number of characters in the converted multibyte character string.
- Returns -1 if 's' does not configure a correct multibyte character string.

mbtowc

Multi-byte Character Multi-byte Character String Manipulate Functions

Function: Converts a multibyte character into a wide character.

Format:
```
#include <stdlib.h>

int mbtowc(wcs, s, n);
```

Method: function

Argument:

wchar_t _far *wcs;	Pointer to an area for storing conversion wide character string
const char _far *s;	Pointer to a multibyte character string
size_t n;	Number of wide characters stored

ReturnValue:

- Returns the number of wide characters converted if 's' configure a correct multibyte character string.
- Returns -1 if 's' does not configure a correct multibyte character string.
- Returns 0 if 's' indicates a NULL character.

memchr

Memory Handling Functions

Function: Searches a character from a memory area.

Format:
```
#include <string.h>

void _far * memchr(s, c, n);
```

Method: function

Argument:

const void _far *s;	Pointer to the memory area to be searched from
int c;	Character to be searched
size_t n;	Size of the memory area to be searched

ReturnValue:

- Returns the position (pointer) of the specified character "c" where it is found.
- Returns NULL if the character "c" could not be found in the memory area.

Description:

- Searches for the characters shown in "c" in the amount of memory specified in "n" starting at the address specified in "s".
- When you specify options -O[3-5], -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 _far *s2;	Pointer to the second memory area to be compared
size_t n;	Number of bytes to be compared

ReturnValue:

- Return Value==0 — The two memory areas are equal.
- Return Value>0 — The first memory area (s1) is greater than the other.
- Return Value<0 — The second memory area (s2) is greater than the other.

Description:

- Compares each of n bytes of two memory areas
- When you specify options -O[3-5], -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);
```

Method: macro(default) or function

Argument:

void _far *s1;	Pointer to the memory area to be copied to
const void _far *s2;	Pointer to the memory area to be copied from
size_t n;	Number of bytes to be copied

ReturnValue: Returns the pointer to the memory area to which the characters have been copied.

Description:

- Usually, the program code described by macro is used for this function. In using the function in a library, please describe it as #undef memcpy after description of #include <string.h>.
- Copies "n" bytes from memory "S2" to memory "S1".
- When you specify options -O[3-5], -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[3-5], -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.

Description: When you specify options -O[3-5], -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>

void _far * memset( s, c, n);
```

Method: macro or function

Argument:

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

ReturnValue: Returns the pointer to the memory area which has been set.

Description:

- Usually, the program code described by macro is used for this function. In using the function in a library, please describe it as #undef memset after description of #include <string.h>.
- Sets "n" bytes of data "c" in memory "s".
- When you specify options -O[3-5], -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.

P

perror

Input/Output Functions

Function: Outputs an error message to stderr.

Format:
```
#include <stdio.h>
void perror(s);
```

Method: function

Argument: const char _far *s; Pointer to a character string attached before a message.

ReturnValue: No value is returned.

pow

Mathematical Functions

Function: Calculates the power of a number.

Format:
```
#include <math.h>
double pow(x, y);
```

Method: function

Argument:

double x;	multiplicand
double y;	power of a numbe

ReturnValue: Returns the multiplicand “x” raised to the power of “y.”

printf

Input/Output Functions

Function: Outputs characters with format to stdout.

Format:
```
#include <stdio.h>

int printf(format, argument...);
```

Method: function

Argument: const char _far *format; Pointer of the format specifying character string

The part after the percent (%) sign in the character string given in format has the following meaning. The part between [and] is optional. Details of the format are shown below.

> Format: %[flag][minimum field width][precision][modifier (l, L, or h)] conversion specification character
>
> Example format: %-05.8ld

ReturnValue:
- Returns the number of characters output.
- Returns EOF if a hardware error occurs.

Description:
- Converts argument to a character string as specified in format and outputs the character string to stdout.
- When giving a pointer to argument, it is necessary to be a far type pointer.

(1) Conversion specification symbol

- d, i
  Converts the integer in the parameter to a signed decimal.
- u
  Converts the integer in the parameter to an unsigned decimal.
- o
  Converts the integer in the parameter to an unsigned octal.
- x
  Converts the integer in the parameter to an unsigned hexadecimal. Lowercase "abcdef" are equivalent to 0AH to 0FH.
- X
  Converts the integer in the parameter to an unsigned hexadecimal. Uppercase "ABCDEF" are equivalent to 0AH to 0FH.
- c
  Outputs the parameter as an ASCII character.
- s
  Converts the parameter after the string far pointer (char *) (and up to a null character '/0' or the precision) to a character string. Note that wchar_t type character strings cannot be processed.[1]
- p
  Outputs the parameter pointer (all types) in the format 24 bits address.
- n
  Stores the number of characters output in the integer pointer of the parameter. The parameter is not converted.

[1] In the standard library included with your product, the character string pointer is a far pointer. (All printf functions handle %s with a far pointer.) Note that scanf functions use a near pointer by default.

printf

Input/Output Functions

Description:

- e
  Converts a double-type parameter to the exponent format. The format is [-]d.dddddde±dd.
- E
  Same as e, except that E is used in place of e for the exponent.
- f
  Converts double parameters to [-]d.dddddd format.
- g
  Converts double parameters to the format specified in e or f. Normally, f conversion, but conversion to e type when the exponent is -4 or less or the precision is less than the value of the exponent.
- G
  Same as g except that E is used in place of e for the exponent.
- −
  Left-aligns the result of conversion in the minimum field width. The default is right alignment.
- +
  Adds + or − to the result of signed conversion. By default, only the - is added to negative numbers.
- Blank' '
  By default, a blank is added before the value if the result of signed conversion has no sign.
- #
  Adds 0 to the beginning of o conversion.
  Adds 0x or 0X to the beginning when other than 0 in x or X conversion.
  Always adds the decimal point in e, E, and f conversion.
  Always adds the decimal point in g and G conversion and also outputs any 0s in the decimal place.

(2) Minimum field width

- Specifies the minimum field width of positive decimal integers.
- When the result of conversion has fewer characters than the specified field width, the left of the field is padded.
- The default padding character is the blank. However, '0' is the padding character if you specified the field with using an integer preceded by '0'.
- If you specified the − flag, the result of conversion is left aligned and padding characters (always blanks) inserted to the right.
- If you specified the asterisk (*) for the minimum field width, the integer in the parameter specifies the field width. If the value of the parameter is negative, the value after the −flag is the positive field width.

(3) Precision

Specify a positive integer after '.'. If you specify only '.' with no value, it is interpreted as zero. The function and default value differs according to the conversion type.
Floating point type data is output with a precision of 6 by default. However, no decimal places are output if you specify a precision of 0.

- d, i, o, u, x, and X conversion
  (1) If the number of columns in the result of conversion is less than the specified number, the beginning is padded with zeros.
  (2) If the specified number of columns exceeds the minimum field width, the specified number of columns takes precedence.

printf

Input/Output Functions

Description:

(3) If the number of columns in the specified precision is less than the minimum field width the field width is processed after the minimum number of columns have bee processed.
(4) The default is 1
(5) Nothing is output if zero with converted by zero minimum columns.

- s conversion
  (1) Represents the maximum number of characters.
  (2) If the result of conversion exceeds the specified number of characters, the remainder is discarded.
  (3) There is no limit to the number of characters in the default.
  (4) If you specify an asterisk (*) for the precision, the integer of the parameter specifies the precision.
  (5) If the parameter is a negative value, specification of the precision is invalid.
- e, E, and f conversion
  n (where n is the precision) numerals are output after the decimal point.
- g and G conversion
  Valid characters in excess of n (where n is the precision) are not output.

(4) I, L[1] 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] In the standard C specifications, variables e, E, f, and g conversions are performed in the case of L on long double parameters. In NC100, long double types are processed as double types.Threfore, if you specify L, the parameters are processed as double types.

putc

Input/Output Functions

Function: Outputs one character to the stream.

Format:
```
#include <stdio.h>

int putc(c, stream);
```

Method: macro

Argument:

int c;	Character to be output
FILE _far *stream;	Pointer of the stream

ReturnValue:
- Returns the output character if output normally.
- Returns EOF if an error occurs.

Description: Outputs one character to the stream.

putchar

Input/Output Functions

Function: Outputs one character to stdout.

Format:
```
#include <stdio.h>

int putchar(c);
```

Method: macro

Argument:

int c;	Character to be output

ReturnValue:
- Returns the output character if output normally.
- Returns EOF if an error occurs.

Description: Outputs one character to stdout.

puts

Input/Output Functions

Function: Outputs one line to stdout.

Format:
```
#include <stdio.h>

int puts(str);
```

Method: macro

Argument: char _far *str; Pointer of the character string to be output

ReturnValue:
- Returns 0 if output normally.
- Returns -1 (EOF) if an error occurs.

Description:
- Outputs one line to stdout.
- The null character ('¥0') at the end of the character string is replaced with the new line character ('¥n').

Q

qsort

Integer Arithmetic Functions

Function: Sorts elements in an array.

Format:
```
#include <stdlib.h>

void qsort(base, nelen, size, cmp(e1, e2));
```

Method: function

Argument:

void _far *base;	Start address of array
size_t nelen;	Element number
size_t size;	Element size
int cmp();	Compare function

ReturnValue: No value is returned.

Description: Sorts elements in an array.

R

rand

Integer Arithmetic Functions

Function: Generates a pseudo-random number.

Format:
```
#include <stdlib.h>

int rand(void);
```

Method: function

Argument: No argument used.

ReturnValue:
- Returns the seed random number series specified in srand.
- The generated random number is a value between 0 and RAND_MAX.

realloc

Memory Management Functions

Function: Changes the size of an allocated memory area.

Format:
```
#include <stdlib.h>

void _far * realloc(cp, nbytes);
```

Method: function

Argument:

void _far *cp;	Pointer to the memory area before change
size_t nbytes;	Size of memory area (in bytes) to be changed

ReturnValue:
- Returns the pointer of the memory area which has had its size changed.
- Returns NULL if a memory area of the specified size could not be secured.

Description:
- Changes the size of an area already secured using malloc or calloc.
- Specify a previously secured pointer in parameter "cp" and specify the number of bytes to change in "nbytes".

S

scanf

Input/Output Functions

Function: Reads characters with format from stdin.

Format:

```
#include <stdio.h>
#include <ctype.h>

int scanf(format, argument...);
```

Method: function

Argument: const char _far *format; Pointer of format specifying character string

The part after the percent (%) sign in the character string given in format has the following meaning. The part between [and] is optional. Details of the format are shown below.

> Format:
> %[*][maximum field width] [modifier (I, L, or h)]conversion specification character
> Example format: %*5ld

ReturnValue:

- Returns the number of data entries stored in each argument.
- Returns EOF if EOF is input from stdin as data.

Description:

- Converts the characters read from stdin as specified in format and stores them in the variables shown in the arguments.
- Argument must be a far pointer to the respective variable.
- The first space character is ignored except in c and [] conversion.
- Interprets code 0x1A as the end code and ignores any subsequent data.

scanf

Input/Output Functions

Description:

(1) Conversion specification symbol

- d
  Converts a signed decimal. The target parameter must be a pointer to an integer.
- i
  Converts signed decimal, octal, and hexadecimal input. Octals start with 0. Hexadecimals start with 0x or 0X. The target parameter must be a pointer to an integer.
- u
  Converts an unsigned decimal. The target parameter must be a pointer to an unsigned integer.
- o
  Converts a signed octal. The target parameter must be a pointer to an integer.
- x, X
  Converts a signed hexadecimal. Uppercase or lowercase can be used for 0AH to 0FH. The leading 0x is not included. The target parameter must be a pointer to an integer.
- s
  Stores character strings ending with the null character '¥0'. The target parameter must be a pointer to a character array of sufficient size to store the character string including the null character '¥0'.
  If input stops when the maximum field width is reached, the character string stored consists of the characters to that point plus the ending null character.
- c
  Stores a character. Space characters are not skipped. If you specify 2 or more for the maximum field width, multiple characters are stored. However, the null character '¥0' is not included. The target parameter must be a pointer to a character array of sufficient size to store the character string.
- p
  The pointer of the argument is output.
- [ ]
  Stores the input characters while the one or more characters between [and] are input. Storing stops when a character other than those between [and] is input. If you specify the circumflex (^) after [, only character other than those between the circumflex and] are legal input characters. Storing stops when one of the specified characters is input.
  The target parameter must be a pointer to a character array of sufficient size to store the character string including the null character '¥0', which is automatically added.
- n
  Stores the number of characters already read in format conversion. The target parameter must be a pointer to an integer.
- e, E, f, g, G
  Convert to floating point format. If you specify modifier I, the target parameter must be a pointer to a double type. The default is a pointer to a float type.

scanf

Input/Output Functions

Description:

(2) *(prevents data storage)

- Specifying the asterisk (*) prevents the storage of converted data in the parameter.

(3) Maximum field width

- Specify the maximum number of input characters as a positive decimal integer. In any one format conversion, the number of characters read will not exceed this number.
- If, before the specified number of characters has been read, a space character (a character that is true in function isspace()) or a character other than in the specified format is input, reading stops at that character.

(4) I, L or h

- I: The results of d, i, o, u, and x conversion are stored as long int and unsigned long int. The results of e, E, f, g, and G conversion are stored as double.
- h: The results of d, i, o, u, and x conversion are stored as short int and unsigned short int.
- If I or h are specified in other than d, i, o, u, or x conversion, they are ignored.
- L: The results of e, E, f, g, and G conversion are stored as float.

setjmp

Execution Control Functions

Function: Saves the environment before a function call

Format:
```
#include <setjmp.h>
int setjmp(env);
```

Method: function

Argument: jmp_buf env; Pointer to the area where environment is saved

ReturnValue: Returns the numeric value given by the argument of longjmp.

Description: Saves the environment to the area specified in "env".

setlocale

Localization Functions

Function: Sets and searches the locale information of a program.

Format:
```
#include <locale.h>
char _far *setlocale(category, locale);
```

Method: function

Argument:

int category;	Locale information, search section information
const char _far *locale;	Pointer to a locale information character string

ReturnValue:

- Returns a pointer to a locale information character string.
- Returns NULL if information cannot be set or searched.

sin

Mathematical Functions

Function: Calculates sine.

Format:
```
#include <math.h>
double sin(x);
```

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the sine of given real number “x” handled in units of radian.

sinh

Mathematical Functions

Function: Calculates hyperbolic sine.

Format:
```
#include <math.h>
double sinh(x);
```

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the hyperbolic sine of given real number "x".

sprintf

Input/Output Functions

Function: Writes text with format to a character string.

Format:
```
#include <stdio.h>
int sprintf(pointer, format, argument...);
```

Method: function

Argument:

char _far *pointer;	Pointer of the location to be stored
const char _far *format;	Pointer of the format specifying character string

ReturnValue: Returns the number of characters output.

Description:

- Converts argument to a character string as specified in format and stores them from the pointer.
- Format is specified in the same way as in printf.

sqrt

Mathematical Functions

Function: Calculates the square root of a numeric value.

Format:
```
#include <math.h>
double sqrt(x);
```

Method: function

Argument: double x; arbitrary real number

ReturnValue: Returns the square root of given real number "x".

srand

Integer Arithmetic Functions

Function: Imparts seed to a pseudo-random number generating routine.

Format:

```
#include <stdlib.h>

void srand(seed);
```

Method: function

Argument: unsigned int seed; Series value of random number

ReturnValue: No value is returned.

Description: Initializes (seeds) the pseudo random number series produced by rand using seed.

sscanf

Input/Output Functions

Function: Reads data with format from a character string.

Format:

```
#include <stdio.h>

int sscanf(string, format, argument...);
```

Method: function

Argument:

const char _far *string;	Pointer of the input character string
const char _far *format;	Pointer of the format specifying character string

ReturnValue:

- Returns the number of data entries stored in each argument.
- Returns EOF if null character ('¥0') is input as data.

Description:

- Converts the characters input as specified in format and stores them in the variables shown in the arguments.
- Argument must be a far pointer to the respective variable.
- Format is specified in the same way as in scanf.

strcat

String Handling Functions

Function: Concatenates character strings.

Format:
```
#include <string.h>

char _far * strcat(s1, s2);
```

Method: function

Argument:

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

ReturnValue: Returns a pointer to the concatenated character string area (s1).

Description:
- Concatenates character strings "s1" and "s2" in the sequence s1+s2[1]
- The concatenated string ends with NULL.
- When you specify options -O[3-5], -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[3-5], -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: Macro, function

Argument:

const char _far *s1;	Pointer to the first character string to be compared
const char _far *s2;	Pointer to the second character string to be compared

ReturnValue:

- ReturnValue== 0 The two character strings are equal.
- ReturnValue>0 The first character string (s1) is greater than the other.
- ReturnValue<0 The second character string (s2) is greater than the other.

Description:

- Usually, the program code described by macro is used for this function. In using the function in a library, please describe it as #undef strcmp after description of #include <string.h>.
- Compares each byte of two character strings ending with NULL
- When you specify options -O[3-5], -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[3-5] 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: macro or function

Argument:

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

ReturnValue: Returns a pointer to the character string at the destination of copy.

Description:

- Usually, the program code described by macro is used for this function. In using the function in a library, please describe it as #undef strcpy after description of #include <string.h>.
- Copies character string "s2" (ending with NULL) to area "s1"
- After copying, the character string ends with NULL.
- When you specify options -O[3-5], -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.

Description: stderr returns the pointer for a static array.

strlen

String Handling Functions

Function: Calculates the number of characters in a character string.

Format:
```
#include <string.h>

size_t strlen(s);
```

Method: function

Argument: const char _far *s; Pointer to the character string to be operated on to calculate length

ReturnValue: Returns the length of the character string.

Description: Determines the length of character string "s" (to NULL).

strncat

String Handling Functions

Function: Concatenates character strings ('n' characters).

Format:
```
#include <string.h>

char _far * strncat(s1, s2, n);
```

Method: function

Argument:

char _far *s1;	Pointer to the character string to be concatenated to
const char _far *s2;	Pointer to the character string to be concatenated from
size_t n;	Number of characters to be concatenated

ReturnValue: Returns a pointer to the concatenated character string area.

Description:

- Concatenates character strings "s1" and "n" characters from character string "s2".
- The concatenated string ends with NULL.
- When you specify options -O[3-5], -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[3-5], -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[3-5], -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[3-5], -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[3-5], -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[3-5], -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 character string.

Format:
```
#include <string.h>

size_t strspn(s1, s2);
```

Method: function

Argument:

const char _far *s1;	Pointer to the character string to be searched in
const char _far *s2;	Pointer to the character string of the character to be searched for

ReturnValue:

- Returns the length (number) of specified characters.

Description:

- Calculates the size of the first character string consisting of characters in “s2” from area “s1”, and searches the characters from the beginning of 's1'.
- You cannot search for '¥0'.
- When you specify options -O[3-5], -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[3-5], -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 * _far *endptr;	Pointer to the remaining character strings that have not been converted

ReturnValue:
- ReturnValue ==0L Does not constitute a number.
- ReturnValue != 0L Returns the configured number in double type.

Description: When you specify options -O[3-5], -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:
- In the first call, returns a pointer to the first character of the first token. A NULL character is written after the returned character. In subsequent calls (when "s1" is NULL), this instruction returns each token as it is encountered. NULL is returned when there are no more tokens in "s1".
- When you specify options -O[3-5], -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[3-5], -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[3-5], -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[3-5], -OR, or -OS, the system may selects another functions with good code efficiency by optimization.

T

tan

Mathematical Functions

Function:	Calculates tangent.	
Format:	`#include <math.h>`	
	`double tan(x);`	
Method:	function	
Argument:	double x;	arbitrary real number
ReturnValue:	Returns the tangent of given real number "x" handled in units of radian.	

tanh

Mathematical Functions

Function:	Calculates hyperbolic tangent.	
Format:	`#include <math.h>`	
	`double tanh(x);`	
Method:	function	
Argument:	double x;	arbitrary real number
ReturnValue:	Returns the hyperbolic tangent of given real number "x".	

tolower

Character Handling Functions

Function:	Converts the character from an upper-case to a lower-case.	
Format:	`#include <ctype.h>`	
	`int tolower(c);`	
Method:	macro	
Argument:	int c;	Character to be converted
ReturnValue:	● Returns the lower-case letter if the argument is an upper-case letter. ● Otherwise, returns the passed argument as is.	
Description:	Converts the character from an upper-case to a lower-case.	

toupper

Character Handling Functions

Function: Converts the character from a lower-case to an upper-case.

Format:
```
#include <ctype.h>

int toupper(c);
```

Method: macro

Argument: int c; Character to be converted

ReturnValue:
- Returns the upper-case letter if the argument is a lower-case letter.
- Otherwise, returns the passed argument as is.

Description: Converts the character from a lower-case to an upper-case.

U

ungetc

Input/Output Functions

Function: Returns one character to the stream

Format:
```
#include <stdio.h>

int ungetc(c, stream);
```

Method: macro

Argument:

int c;	Character to be returned
FILE _far *stream;	Pointer of stream

ReturnValue:
- Returns the returned one character if done normally.
- Returns EOF if the stream is in write mode, an error or EOF is encountered, or the character to be sent back is EOF.

Description:
- Returns one character to the stream.
- Interprets code 0x1A as the end code and ignores any subsequent data.

V

vfprintf

Input/Output Functions

Function: Output to a stream with format.

Format:
```
#include <stdarg.h>
#include <stdio.h>

int vfprintf(stream, format, ap...);
```

Method: function

Argument:

FILE _far *stream;	Pointer of stream
const char _far *format;	Pointer of the format specifying character string
va_list ap;	Pointer of argument list

ReturnValue: Returns the number of characters output.

Description:

- Output to a stream with format.
- When writing pointers in variable-length variables, make sure they are a far-type pointer.

vprintf

Input/Output Functions

Function: Output to stdout with format.

Format:
```
#include <stdarg.h>
#include <stdio.h>

int vprintf(format, ap...);
```

Method: function

Argument:

const char _far *format;	Pointer of the format specifying character string
va_list ap;	Pointer of argument list

ReturnValue: Returns the number of characters output.

Description:

- Output to stdout with format.
- When writing pointers in variable-length variables, make sure they are a far-type pointer.

vsprintf

Input/Output Functions

Function: Output to a buffer with format.

Format:

```
#include <stdarg.h>
#include <stdio.h>

int vfprintf(s, format, ap...);
```

Method: function

Argument:

char _far *s;	Pointer of the location to be store
const char _far *format;	Pointer of the format specifying character string
va_list ap;	Pointer of argument list

ReturnValue: Returns the number of characters output.

Description: When writing pointers in variable-length variables, make sure they are a far-type pointer.

W

wcstombs

Multi-byte Character Multi-byte Character String Manipulate Functions

Function: Converts a wide character string into a multibyte character string.

Format:
```
#include <stdlib.h>

size_t _far wcstombs(s, wcs, n);
```

Method: function

Argument:

char _far *s;	Pointer to an area for storing conversion multibyte character string
const wchar_t _far *wcs;	Pointer to a wide character string
size_t n;	Number of wide characters stored

ReturnValue:
- Returns the number of stored multibyte characters if the character string was converted correctly.
- Returns -1 if the character string was not converted correctly.

wctomb

Multi-byte Character Multi-byte Character String Manipulate Functions

Function: Converts a wide character into a multibyte character.

Format:
```
#include <stdlib.h>

int wctomb(s, wchar);
```

Method: function

Argument:

char _far *s;	Pointer to an area for storing conversion multibyte character string
wchar_t wchar;	wide character

ReturnValue:
- Returns the number of bytes contained in the multibyte characters.
- Returns -1 if there is no corresponding multibyte character.
- Returns 0 if the wide character is 0.

E.2.4 Using the Standard Library

a Notes on Regarding Standard Header File

When using functions in the standard library, always be sure to include the specified standard header file. If this header file is not included, the integrity of arguments and return values will be lost, making the program unable to operate normally.

b Notes on Regarding Optimization of Standard Library

If you specify any of optimization options -O[3-5], -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 inTable E.13 are met.

Table E.13 Optimization Conditions for Standard Library Functions

Function Name	Optimization Condition	Description Example
strcpy	First argument:far pointer Second argument:string constant	strcpy(str, "sample");
memcpy	First argument:far pointer Second argument: far pointer Third argument:constant	memcpy(str ,"sample", 6); memcpy(str , fp, 6);

E.3 Modifying Standard Library

The NC100 package includes a sophisticated function library which includes functions such as the scanf and printf I/O functions. These functions are normally called high-level I/ O functions. These high-level I/O functions are combinations of hardware-dependent lowlevel I/O functions.
In R32C/100 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 NC100 standard library to match the target system.
The entry vedrsion does not come with source files for the standard function library. Therefore, the standard function library cannot be customized for the entry version.

E.3.1 Structure of I/O Functions

As shown in Figure E.1, the I/O functions work by calling lower-level functions (level 2 . level 3) from the level 1 function. For example, fgets calls level 2 fgetc, and fgetc calls a level 3 function.
Only the lowest level 3 functions are hardware-dependent (I/O port dependent) in the Micro Processor. If your application program uses an I/O function, you may need to modify the source files for the level 3 functions to match the system.

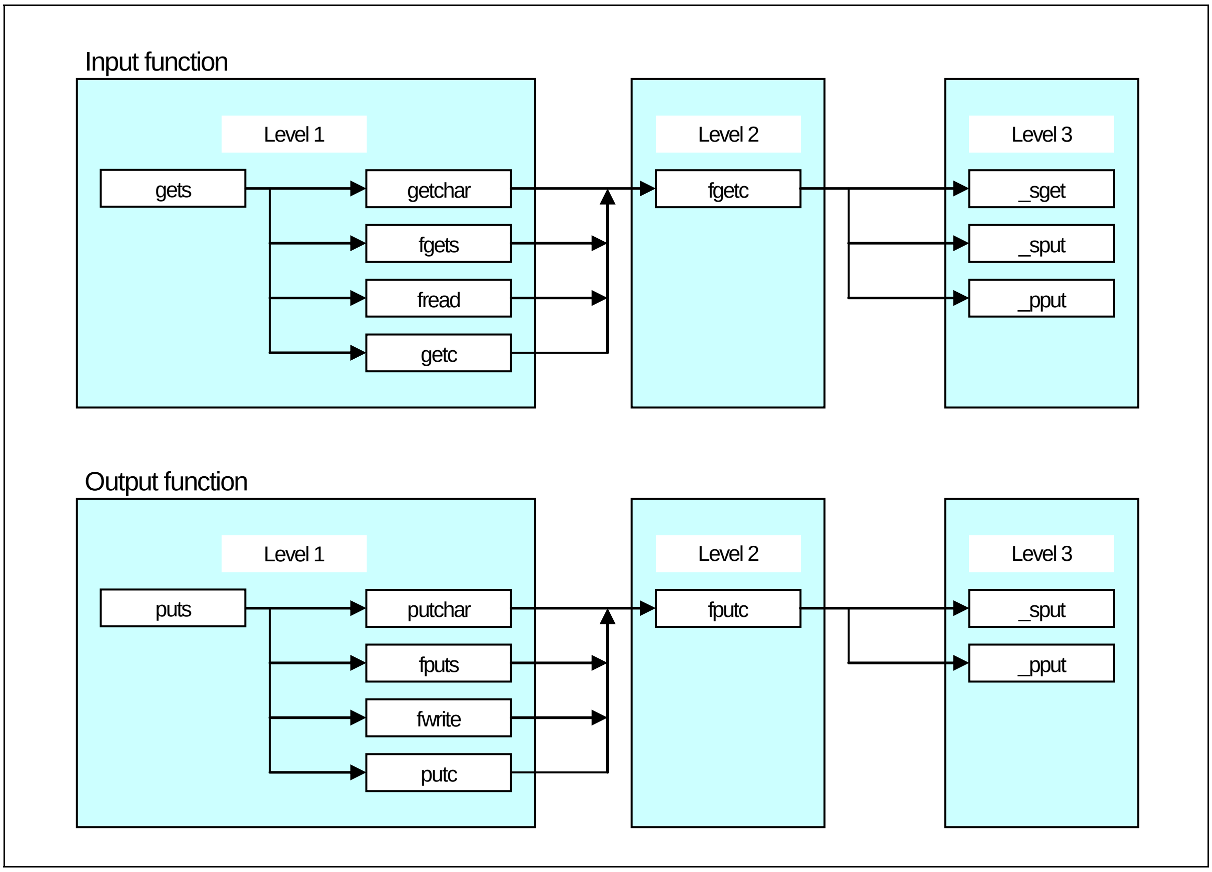


Figure E.1 Calling Relationship of I/O Functions

E.3.2 Sequence of Modifying I/O Functions

Figure E.2 outlines how to modify the I/O functions to match the target system.

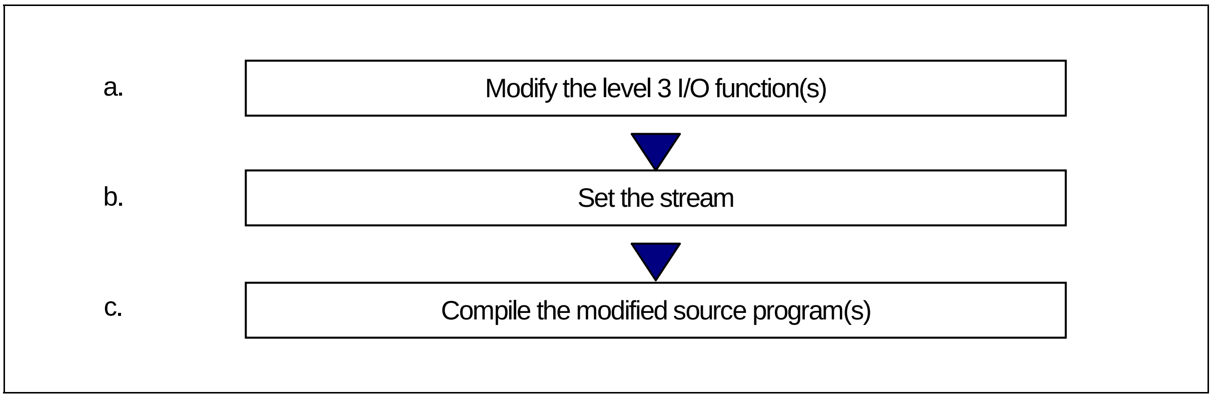


Figure E.2 Example Sequence of Modifying I/O Functions

a Modifying Level 3 I/O Function

The level 3 I/O functions perform 1-byte I/O via the R32C/100 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.

(1) Circuit settings

- Clock frequency: 20MHz

(2) Initial serial communications settings

- Use UART1
- Baud rate: 9600bps
- Data size: 8 bits
- Parity: None
- Stop bits: 2 bits

*The initial serial communications settings are made in the init function (init.c).

The level 3 I/O functions are written in the C library source file device.c. Table E.14 lists the specifications of these functions.

Table E.14 Specifications of Level 3 Functions

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

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

Serial communication is set to UART1 in the R32C/100 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 NC100 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 NC100 standard library functions shown shaded in Figure E.3 are fixed to standard input (stdin) and standard output (stdout). The stream cannot be changed for these functions. The output direction of stderr is defined as stdout in #define.

The stream can only be changed for functions that specify pointers to the stream as parameters such as fgetc and fputc.

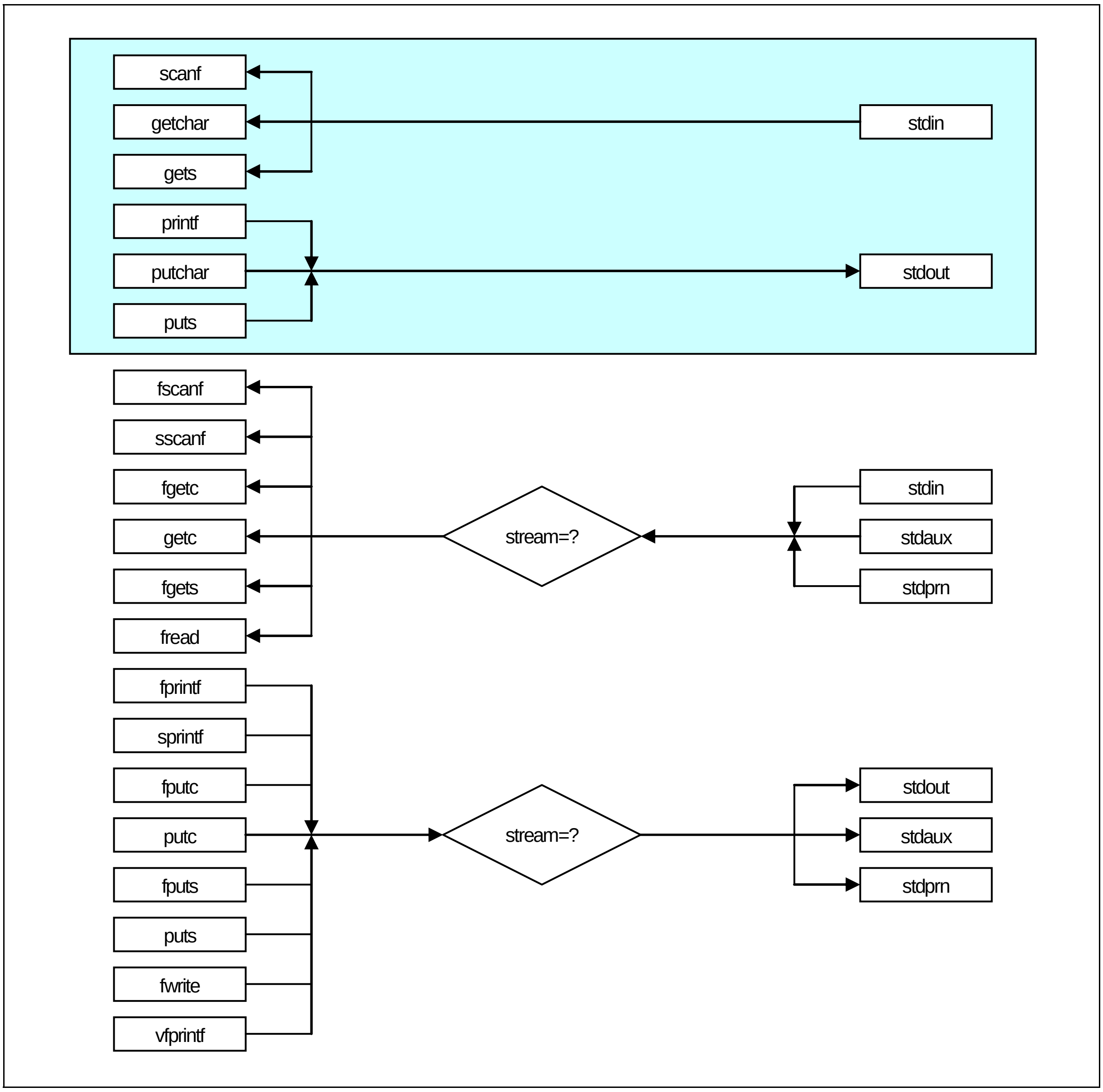


Figure E.3 fRelationship of Functions and Streams

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

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

#define     stderr      stdout              /* NC no-support */

/*******************************************************
*
*******************************************************/
#define     _IOREAD     1           /* Read only flag */
#define     _IOWRT      2           /* Write only flag */
#define     _IOEOF      4           /* End of file flag */
#define     _IOERR      8           /* Error flag */
#define     _IORW       16          /* Read and write flag */
#define     _NFILE      4           /* Stream number */
#define     _TEXT       1           /* Text mode flag */
#define     _BIN        2           /* Binary mode flag */

            (remainder omitted)
                :
```

Figure E.4 Stream Definition in stdio.h

Let's look at the elements of the file structures shown in Figure E.4. Items [1] to [6] correspond to [1] to [6] in Figure E.4

(1) char _buff

Functions scanf and fscanf read one character ahead during input. If the character is no use, function ungetc is called and the character is stored in this variable.

If data exists in this variable, the input function uses this data as the input data.

(2) int _cnt

Stores the _buff data count (0 or 1)

(3) int _flag

Stores the read-only flag (_IOREAD), the write-only flag (_IOWRT), the read-write flag (_IORW), the end of file flag (_IOEOF) and the error flag (_IOERR).

- _IOREAD, _IOWRT, _IORW

  These flags specify the stream operating mode. They are set during stream initialization.

- _IOEOF, _IOERR

  These flags are set according to whether an EOF is encountered or error occurs in the I/O function.

(4) int _mod

Stores the flags indicating the text mode (_TEXT) and binary mode (_BIN).

- Text mode

  Echo-back of I/O data and conversion of characters. See the source programs (fgetc.c and fputc.c) of the fgetc and fputc functions for details of echo back and character conversion.

- Binary mode

  No conversion of I/O data. These flags are set in the initialization block of the stream.

(5) int (*_func_in)()

When the stream is in read-only mode (_IOREAD) or read/write mode (_IORW), stores the level 3 input function pointer. Stores a NULL pointer in other cases.

This information is used for indirect calling of level 3 input functions by level 2 input functions.

(6) int (*_func_out)()

When the stream is in write mode (_IOWRT), stores the level 3 output function pointer. If the stream can be input (_IOREAD or _IORW), and is in text mode, it stores the level 3 output function pointer for echo back. Stores a NULL pointer in other cases.

This information is used for indirect calling of level 3 output functions by level 2 output functions.

Set values for all elements other than char_buff in the stream initialization block. The standard library file supplied in the NC100 package initializes the stream in function init, which is called from the ncrt0.a30 startup program.
Figure E.5 shows the source program for the init function.

```
#include <stdio.h>

FILE _iob[4];

void init( void );

void init( void )
{
        stdin->_cnt = stdout->_cnt = stdaux->_cnt = stdprn->_cnt = 0;
        stdin->_flag = _IOREAD;
        stdout->_flag = _IOWRT;
        stdaux->_flag = _IORW;
        stdprn->_flag = _IOWRT;

        stdin->_mod = _TEXT;
        stdout->_mod = _TEXT;
        stdaux->_mod = _BIN;
        stdprn->_mod = _TEXT;

        stdin->_func_in = _sget;
        stdout->_func_in = NULL;
        stdaux->_func_in = _sget;
        stdprn->_func_in = NULL;

        stdin->_func_out = _sput;
        stdout->_func_out = _sput;
        stdaux->_func_out = _sput;
        stdprn->_func_out = _pput;

#ifdef UART0
        speed(_96, _B8, _PN, _S2);
#else /* UART1 : default */
        speed(_96, _B8, _PN, _S2);
#endif
        init_prn();
}
```

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

In systems using the two R32C/100 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)
                :
    speed(_96, _B8, _PN, _S2);          ← [4](set UART0 speed function)
    init_prn();
}

* [2] to [4] correspond to the items in the description of setting, above.
```

Figure E.6 Modifying the init Function

c Incorporating the Modified Source Program

There are two methods of incorporating the modified source program in the target system:

(1) Specify the object files of the modified function source files when linking.

(2) Use the makefile (under MS-Windows, makefile.dos) supplied in the NC100 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).

```
% nc100  -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 NC100, and their countermeasures.

F.1 Message Format

If, during processing, NC100 detects an error, it displays an error message on the screen and stops the compiling process.
Figure F.1 to Figure F.3 shows the format of error messages and warning messages.

```
nc100:[error-message]
```

Figure F.1 Format of Error Messages from the nc100 Compile Driver

```
[Error(cpp100.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(cpp100. warning-No.): filename, line-No.] warning-message
[Warning(ccom): filename, line-No.] warning-message
```

Figure F.3 Format of Command Warning Messages

*1. Fatal error message
This error message is not normally output. Please contact nearest Renesas office. with details of the message if displayed.

F.2 nc100 Error Messages

Table F.1 and Table F.2 list the nc100 compile driver error messages and their countermeasures.

Table F.1 nc100 Error Messages (1/2)

Error message	Description and countermeasure
Arg list too long	• The command line for starting the respective processingsystem is longer than the character string defined bythe system. ⇒ Specify a NC100 option to ensure that the number ofcharacters defined by the system is not exceeded. Usethe -v option to check the command line used for eachprocessing block.
Cannot analyze error	• This error message is not normally displayed. (It is aninternal error.) ⇒ Contact Renesas Solutions Corp.
command-file line characters exceed 2048.	• There are more than 2048 characters on one or more lines in the command file. ⇒ Reduce the number of characters per line in the commandfile to 2048 max.
Core dump(command_name)	• The processing system (indicated in parentheses)caused a core dump. ⇒ The processing system is not running correctly. Checkthe environment variables and the directory containingthe processing system. If the processing system stilldoes not run correctly, Please contact Renesas SolutionsCorp.
Exec format error	• Corrupted processing system executable file. ⇒ Reinstall the processing system.
Ignore option '-?'	• You specified an illegal option (-?) for NC100. ⇒ Specify the correct option.
illegal option	• You specified options greater than 100 characters for –as100 or –ln100. ⇒ Reduce the options to 99 characters or less.
Invalid argument	• This error message is not normally displayed. (It is aninternal 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.
Invalid suffix '.xxx'	• You specified a filename extension not recognized by NC100 (other than .c, .i, .a30, .r30, .x30). ⇒ Specify the filename with the correct extension.

Table F.2 nc100 Error Messages (2/2)

Error message	Description and countermeasure
No such file or directory	• The processing system will not run. ⇒ Check that the directory of the processing system is correctly set in the environment variable.
Not enough core	• Insufficient swap area ⇒ Increase the swap area.
Permission denied	• The processing system will not run. ⇒ Check access permission to the processing systems. Or, if access permission is OK, check that the directory of the processing system is correctly set in the environment variable.
can't open command file	• Can not open the command file specified by '@'. ⇒ Specify the correct input file.
too many options	• This error message is not normally displayed. (It is an internal error.) ⇒ Compile options cannot be specified exceeding 99 characters.
Result too large	• 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 cpp100 Error Messages

Table F.3 to Table F.5 list the error messages output by the cpp100 preprocessor and their countermeasures.

Table F.3 cpp100 Error Messages (1/3)

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 cpp100 command line. ⇒ Specify only legal options.
11	cannot open input file.	• Input file not found. ⇒ Specify the correct input file name.
12	cannot close input file.	• Input file cannot be closed. ⇒ Check the input file name.
14	cannot open output file.	• Cannot open output file. ⇒ Specify the correct output file name.
15	cannot close output file.	• Cannot close output file. ⇒ Check the available space on disk.
16	cannot write output file	• Error writing to output file. ⇒ Check the available space on disk.

Table F.4 cpp100 Error Messages (2/3)

No.	Error message	Description and countermeasure
17	input file name buffer overflow	• The input filename buffer has overflowed. Note that the filename includes the path. ⇒ Reduce the length of the filename and path (use the -I option to specify the standard directory).
18	not enough memory for macro include file not found	• Insufficient memory for macro name and contents of macro ⇒ Increase the swap area
21	include file not found	• The include file could not be opened.. ⇒ The include files are in the current directory and that specified in the -I option and environment variable. Check these directories.
22	illegal file name error	• Illegal filename. ⇒ Specify a correct filename.
23	include file nesting over	• Nesting of include files exceeds the limit (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 - #assert operation expression. ⇒ Rewrite operation expression correctly.
27	macro argument error	• Error in number of macro parameters when expanding macro. ⇒ Check macro definition and reference and correct as necessary.
28	input buffer over flow	• Input line buffer overflow occurred when reading source file(s). Or, buffer overflowed when converting macros. ⇒ Reduce each line in the source file to a maximum of 1023 characters. If you anticipate macro conversion, modify the code so that no line exceeds 1023 characters after conversion.
29	EOF in comment	• End of file encountered in a comment. ⇒ Correct the source file.
31	EOF in preprocess command	• End of file encountered in a preprocess command ⇒ Correct the source file.
32	unknown preprocess command	• An unknown preprocess command has been specified. ⇒ Only the following preprocess commands can be used in CPP100： #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 the #include.

Table F.5 cpp100 Error Messages (3/3)

No.	Error message	Description and countermeasure
43	illegal id name	• Error in following macro name or argument in #define command: __FILE__, __LINE__, __DATE__, __TIME__ ⇒ Correct the source file.
44	token buffer over flow	• Token character buffer of #define overflowed. ⇒ Reduce the number of token characters.
45	illegal undef command usage	• Error in #undef. ⇒ Correct the source file.
46	undef id not found	• The following macro names to be undefined in #undef were not defined: __FILE__, __LINE__, __DATE__, __TIME__ ⇒ Check the macro name.
52	illegal ifdef / ifndef command usage	• Error in #ifdef. ⇒ Correct the source file.
53	elseif / else sequence erro	• #elseif or #else were used without #if - #ifdef - #ifndef. ⇒ Use #elseif or #else only after #if - #ifdef -#ifndef.
54	endif not exist	• No #endif to match #if - #ifdef - #ifndef. ⇒ Add #endif to the source file.
55	endif sequence error	• #endif was used without #if - #ifdef - #ifndef. ⇒ Use #endif only after #if - #ifdef - #ifndef.
61	illegal line command usage	• Error in #line. ⇒ Correct the source file.

F.4 cpp100 Warning Messages

Table F.6 shows the warning messages output by cpp100 and their countermeasures.

Table F.6 cpp100 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 cpp100: __FILE__, __LINE__, __DATE__, __TIME__ ⇒ Use a different macro name.
82	assertion warning	• The result of an #assert operation expression was 0. ⇒ Check the operation expression.
83	garbage argument	• Characters other than a comment exist after a preprocess command. ⇒ Specify characters as a comment (/* string */) after the preprocess command.
84	escape sequence out of range for character	• An escape sequence in a character constant or character string constant exceeded 255 characters. ⇒ Reduce the escape sequence to within 255 characters.
85	redefined	• A previously defined macro was redefined with different contents. ⇒ Check the contents against those in the previous definition.
87	/* within comment	• A comment includes /*. ⇒ Do not nest comments.
88	Environment variable 'NCKIN' must be 'SJIS' or 'EUC'	• Environment variable 'NCKIN' is not valid. ⇒ Set "SJIS" or "EUC" to NCKIN.
90	'Macro name' in #if is not defined,so it's tereated as 0	• An undefined macro name in #if is used. ⇒ Check the macro definition.

F.5 ccom100 Error Messages

Table F.7 to Table F.18 list the ccom100 compiler error messages and their countermeasures.

Table F.7 ccom100 Error Messages (1/12)

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 PRAGMA-name & function prototype mismatched	• The function specified by #pragma PRAGMA name does not match the contents of argument in prototype declaration. ⇒ Make sure it is matched to the argument in prototype declaration.
#pragma PRAGMA-name's function argument is struct or union	• The struct or union type is specified in the prototype declaration for the function specified by #pragma PRAGMA-name. ⇒ Specify the int or short type, 2-byte pointer type, or enumeration type in the prototype declaration.
#pragma PRAGMA-name must be declared before use	• A function specified in the #pragma PRAGMAname declaration is defined after call for that function. ⇒ Declare a function before calling it.
#pragma BITADDRESS variable is not _Bool type	• The variable spcified by #pragma BITADDRESS is not _Bool type ⇒ Use the _Bool type to declare the variable.
#pragma INTCALL function's argument on stack	• When the body of functions declared in #pragma INTCALL are written in C, the parameters are passed via the stack. ⇒ When the body of functions declared in #pragma INTCALL are written in C, specify the parameters are being passed via the stack.
#pragma PARAMETER function's register not allocated	• A register which is specifed in the function decleared by #pragma PARAMETER can not be allocated. ⇒ Use the correct register.
'const' is duplicate	• const is described more than twice. ⇒ Write the type qualifier correctly.
'far' & 'near' conflict	• far/near is described more than twice. ⇒ Write near/far correctly.
'far' is duplicate	• far is described more than twice. ⇒ Write far correctly.
'near' is duplicate	• near is described more than twice. ⇒ Write near correctly.
'static' is illegal storage class for agument	• An appropriate storage class is used in argument declaration. ⇒ Use the correct storage class.
'volatile' is duplicate	• volatile is described more than twice. ⇒ Write the type qualifier correctly.

Table F.8 ccom100 Error Messages (2/12)

Error message	Description and countermeasure
(can't read C source from filename line number for error message)	• The source line is in error and cannot be displayed. The file indicated by filename cannot be found or the line number does not exist in the file. ⇒ Check whether the file actually exists.
(can't open C source filename for error message)	• The source file in error cannot be opened. ⇒ Check whether the file exists.
argument type given both places	• Argument declaration in function definition overlaps an argument list separately given. ⇒ Choose the argument list or argument declaration for this argument declaration.
array of functions declared	• The array type in array declaration is defined as function. ⇒ Specify scalar type struct/union for the array type.
array size is not constant integer	• The number of elements in array declaration is not a constant. ⇒ Use a constant to describe the number of elements.
asm()'s string must have only 1 $b	• $b is described more than twice in asm statement. ⇒ Make sure that $b is described only once.
asm()'s string must not have more than 3 $$ or $@	• $$ or $@ is described more than thrice in asm statement. ⇒ Make sure that $$ ($@)is described only twice.
auto variable's size is zero	• An array with 0 elements or no elements was declared in the auto area. ⇒ Correct the coding.
bitfield width exceeded	• The bit-field width exceeds the bit width of the data type. ⇒ Make sure that the data type bit width declared in the bit-field is not exceeded.
bitfield width is not constant integer	• The bit width of the bit-field is not a constant. ⇒ Use a constant to write the bit width.
can't get bitfield address by '&' operator	• The bit-field type is written with the & operator. ⇒ Do not use the & operator to write the bit-field type.
can't get inline function's address by '&' operator	• The & operator is written in an inline function. ⇒ Do not use the & operator in an inline function.
can't get size of bitfield	• The bit-field type is written with the sizeof operator. ⇒ Do not use the sizeof operator to write the bitfield type.
can't get void value	• An attempt is made to get void-type data as in cases where the right side of an assignment expression is the void type. ⇒ Check the data type.
can't output to file-name	• The file cannot be wrote ⇒ Check the rest of disk capacity or permission of the file.
can't open file-name	• The file cannot be opened. ⇒ Check the permission of the file.
can't set argument	• The type of an actual argument does not match prototype declaration. The argument cannot be set in a register (argument). ⇒ Correct mismatch of the type.
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.

Table F.9 ccom100 Error Messages (3/12)

Error message	Description and countermeasure
conflict declare of variable-name	• The variable is defined twice with different storage classes each time. ⇒ Use the same storage class to declare a variable twice.
conflict function argument type of variable-name	• The argument list contains the same variable name. ⇒ Change the variable name.
declared register parameter function's body declared	• The function body for the function declared with #pragma PARAMETER is defined in C ⇒ Do not define , in C, the body for such function .
default function argument conflict	• The default value of an argument is declared more than once in prototype declaration. ⇒ Make sure that the default value of an argument is declared only once.
default: is duplicated	• The default value is used more than one time. ⇒ Use only one default within one switch statement.
do while(struct/union) statement	• The struct or union type is used in the expression of the do-while statement. ⇒ Use the scalar type for an expression in the dowhile statement.
do while(void) statement	• The void type is used in the expression of the dowhile statement. ⇒ Use the scalar type for an expression in the dowhile statement.
duplicate frame position defind variable-name	• Auto variable is described more than twice. ⇒ Write the type specifier correctly.
Empty declare	• Only storage class and type specifiers are found. ⇒ Write a declarator.
float and double not have sign	• Specifiers signed/unsigned are described in float or double. ⇒ Write the type specifier correctly.
floating point value overflow	• The floating-point immediate value exceeds the representable range. ⇒ Make sure the value is within the range.
floating type's bitfield	• A bit-field of an invalid type is declared. ⇒ Use the integer type to declare a bit-field.
for(; struct/union;) statement	• The struct or union type is used in the second expression of the for statement. ⇒ Use the scalar type to describe the second expression of the for statement.
for(; void ;) statement	• The 2nd expression of the for statement has void. ⇒ Use the scalar type as the 2nd expression of the for statement.
function initialized	• An initialize expression is described for function declaration. ⇒ Delete the initialize expression.
function member declared	• A member of struct or union is function type ⇒ Write the members correctly.
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.

Table F.10 ccom100Error message(4/12)

Error message	Description and countermeasure
function returning an array	• The type of the return value in function declaration is an array type. ⇒ Change the type to “pointer to function”etc.
handler function called	• The function specified by #pragma HANDLER is called. ⇒ Be careful not to call a handler.
identifier (variable-name) is duplicated	• The variable is defined more than one time. ⇒ Specify variable definition correctly.
if(struct/union) statement	• The struct or union type is used in the expression of the if statement. ⇒ The expression must have scalar type.
if(void) statement	• The void type is used in the expression of the if statement. ⇒ The expression must have scalar type.
illegal storage class for argument, 'inline' ignored	• An inline function is declared in declaration statement within a function. ⇒ Declare it outside a function.
illegal storage class for argument, 'interrupt' ignored	• An interrupt function is declared in declaration statement within a function. ⇒ Declare it outside a function.
incomplete array access	• An attempt is made to reference an array of incomplete. ⇒ Define size of array.
incomplete return type	• An attempt is made to reference an return variable of incomplete type. ⇒ Check return variable.
incomplete struct get by [ ]	• An attempt is made to reference or initialize an array of incomplete structs or unions that do not have defined members. ⇒ Define complete structs or unions first.
incomplete struct member	• An attempt is made to reference an struct member of incomplete . ⇒ Define complete structs or unions first.
incomplete struct initialized	• An attempt is made to initialize an array of incomplete structs or unions that do not have defined members. ⇒ Define complete structs or unions first.
incomplete struct return function call	• An attempt is made to call a function that has as a return value the of incomplete struct or union that does not have defined members. ⇒ Define a complete struct or union first.
incomplete struct / union's member access	• An attempt is made to reference members of an incomplete struct or union that do not have defined members. ⇒ Define a complete struct or union first.
incomplete struct / union(tagname)' s member access	• An attempt is made to reference members of an incomplete struct or union that do not have defined members. ⇒ Define a complete struct or union first.
inline function have invalid argument or return code	• inline function has an invalid argument or an invalid return value. ⇒ Write the argument or an invalid return value correctly.
inline function is called as normal function before	• The function declared in storage class inline is called as an ordinary function. ⇒ Always be sure to define an inline function before using it.

Table F.11 ccom100Error message(5/12)

Error message	Description and countermeasure
inline function's address used	• An attempt is made to reference the address of an inline function. ⇒ Do not use the address of an inline function.
inline function's body is not declared previously	• The body of an inline function is not defined. ⇒ Using an inline function, define the function body prior to the function call.
inline function (function-name) is recursion	• The recursive call of an in line function cannot be carried out. ⇒ Using an inline function, No recursive.
interrupt function called	• The function specified by #pragma INTERRUPT is called. ⇒ Be careful not to call an interrupt handling function.
invalid environment variable: (environment variable -name)	• The variable name specified in the environment variable NCKIN/NCKOUT is specified by other than SJIS and EUC. ⇒ Check the environment variables used.
invalid function default argument	• The default argument to the function is incorrect. ⇒ This error occurs when the prototype declaration of the function with default arguments and those in the function definition section do not match. Make sure they match.
invalid push	• An attempt is made to push void type in function argument, etc. ⇒ The type void cannot be pushed.
invalid ' ? : ' operand	• The ?: operation contains an error. ⇒ Check each expression. Also note that the expressions on the left and right sides of : must be of the same type.
invalid '!=' operands	• The != operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '&&' operands	• The && operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '&' operands	• The & operation contains an error. ⇒ Check the expression on the right side of the operator.
invalid '&=' operands	• The &= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '()' operand	• The expression on the left side of () is not a function. ⇒ Write a function or a pointer to the function in the left-side expression of ().
invalid '*' operands	• If multiplication, the * operation contains an error. If * is the pointer operator, the right-side expressionis not pointer type. ⇒ For a multiplication, check the expressions on the left and right sides of the operator. For a pointer, check the type of the right-side expression.
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.12 ccom100Error message(6/12)

Error message	Description and countermeasure
invalid '+=' operands	• The += operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '-' operands	• The - operator contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '-=' operands	• The -= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '/=' operands	• The /= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '<<' operands	• The << operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '<<=' operands	• The <<= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '<=' operands	• The <= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '=' operand	• The = operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '==' operands	• The == operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '>=' operands	• The >= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '>>' operands	• The >> operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '>>=' operands	• The >>= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid '[ ]' operands	• The left-side expression of [ ] is not array type or pointer type. ⇒ Use an array or pointer type to write the left-side expression of [ ].
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.13 ccom100Error message(7/12)

Error message	Description and countermeasure
invalid '%=' operands	• The %= operation contains an error. ⇒ Check the expressions on the left and right sides of the operator.
invalid ++ operands	• The ++ unary operator or postfix operator contains an error. ⇒ For the unary operator, check the right-side expression. For the postfix operator, check the leftside expression.
invalid -- operands	• The -- unary operation or postfix operation contains an error. ⇒ For the unary operator, check the right-side expression. For the postfix operator, check the leftside expression.
invalid -> used	• The left-side expression of -> is not struct or union. ⇒ The left-side expression of -> must have struct or union.
invalid (? :)'s condition	• The ternary operator is erroneously written. ⇒ Check the ternary operator.
Invalid #pragma OS Extended function interrupt number	• The INT No. in #pragma OS Extended function is invalid. ⇒ Specify correctly.
Invalid #pragma INTCALL interrupt number	• The INT No. in #pragma INTCALL is invalid. ⇒ Specify correctly.
Invalid #pragma SPECIAL special page number (NC30, NC308 only)	• The No. in #pragma SPECIAL is invalid. ⇒ Specify correctly.
invalid CAST operand	• The cast operation contains an error. The void type cannot be cast to any other type; it can neither be cast from the structure or union type nor can it be cast to the structure or union type. ⇒ Write the expression correctly.
invalid asm()'s argument	• The variables that can be used in asm statements are only the auto variable and argument. ⇒ Use the auto variable or argument for the statement.
invalid bitfield declare	• The bit-field declaration contains an error. ⇒ Write the declaration correctly.
invalid break statements	• The break statement is put where it cannot be used. ⇒ Make sure that it is written in switch, while, dowhile, and for.
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.

Table F.14 ccom100Error message(8/12)

Error message	Description and countermeasure
invalid function's argument declaration	• The argument of the function is erroneously declared. ⇒ Write it correctly.
invalid function declare	• The function definition contains an error. ⇒ Check the line in error or the immediately preceding function definition.
invalid initializer	• The initialization expression contains an error. This error includes excessive parentheses, many initialize expressions, a static variable in the function initialized by an auto variable, or a variable initialized by another variable. ⇒ Write the initialization expression correctly.
invalid initializer of variable-name	• The initialization expression contains an error.This error includes a bit-field initialize expression described with variables, for example. ⇒ Write the initialization expression correctly.
invalid initializer on array	• The initialization expression contains an error. ⇒ Check to see if the number of initialize expressions in the parentheses matches the number of array elements and the number of structure members.
invalid initializer on char array	• The initialization expression contains an error. ⇒ Check to see if the number of initialize expressions in the parentheses matches the number of array elements and the number of structure members.
invalid initializer on scalar	• The initialization expression contains an error. ⇒ Check to see if the number of initialize expressions in the parentheses matches the number of array elements and the number of structure members.
invalid initializer on struct	• The initialization expression contains an error. ⇒ Check to see if the number of initialization expressions in the parentheses matches the number of array elements and the number of structure members.
invalid initializer, too many brace	• Too many braces { } are used in a scalar-type initialization expression of the auto storage class. ⇒ Reduce the number of braces { } used.
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.

Table F.15 ccom100Error message(9/12)

Error message	Description and countermeasure
invalid storage class for data	• The storage class is erroneously specified. ⇒ Write it correctly.
invalid struct or union type	• Structure or union members are referenced for the enumerated type of data. ⇒ Write it correctly.
invalid truth expression	• The void, struct, or union type is used in the first expression of a condition expression (?:). ⇒ Use scalar type to write this expression.
invalid type specifier	• The same type specifier is described twice or more as in "int int i;" or an incompatible type specifier is described as in "float int i;." ⇒ Write the type specifier correctly.
invalid type's bitfield	• A bit-field of an invalid type is declared. ⇒ Use the integer type for bit-fields.
invalid type specifier,long long long	• Specifiers "long" are described thrice or more. ⇒ Check the type.
invalid unary '!' operands	• Use of the ! unary operator is illegal. ⇒ Check the right-side expression of the operator.
invalid unary '+' operands	• Use of the + unary operator is illegal. ⇒ Check the right-side expression of the operator.
invalid unary '-' operands	• Use of the - unary operator is illegal. ⇒ Check the right-side expression of the operator.
invalid unary '~' operands	• Use of the ~ unary operator is illegal. ⇒ Check the right-side expression of the operator.
invalid void type	• The void type specifier is used with long or singed. ⇒ Write the type specifier correctly.
invalid void type, int assumed	• The void-type variable cannot be declared. Processing will be continued by assuming it to be the int type. ⇒ Write the type specifier correctly.
invalid size of bitfield	• Get the bitfield size. ⇒ Not write bitfield on this decraration.
invalid switch statement	• The switch statement is illegal. ⇒ Write it correctly.
label label redefine	• The same label is defined twice within one function. ⇒ Change the name for either of the two labels.
long long type's bitfield	• Specifies bitfield by long long type ⇒ Can not specifies bit-fields of long long type.
mismatch prototyped parameter type	• The argument type is not the type declared in prototype declaration. ⇒ Check the argument type.
No #pragma ENDASM	• #pragma ASM does not have matching #pragma ENDASM. ⇒ Write #pragma ENDASM.
No declarator	• The declaration statement is incomplete. ⇒ Write a complete declaration statement.
Not enough memory	• The memory area is insufficient. ⇒ Increase the memory or the swap area.
not have 'long char'	• Type specifiers long and char are simultaneously used. ⇒ Write the type specifier correctly.

Table F.16 ccom100Error message(10/12)

Error message	Description and countermeasure
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.
Sorry, stack frame memory exhaust, max. 128 bytes but now nnn bytes	• A maximum of 128 bytes of parameters can be secured on the stack frame. Currently, nnn bytes have been used. ⇒ Reduce the size or number of parameters.
Sorry, compilation terminated because of these errors in 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.

Table F.17 ccom100Error message(11/12)

Error message	Description and countermeasure
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.
too big data-length	• An attempt is made to get an address exceeding the 32-bit range. ⇒ Make sure the set values are within the address range of the microcomputer used.
too big address	• An attempt is made to set an address exceeding the 32-bit range. ⇒ Make sure the set values are within the address range of the microcomputer used.
too many storage class of typedef	• Storage class specifiers such as extern/typedef/ static/auto/register are described more than twice in declaration. ⇒ Do not describe a storage class specifier more than twice.
type redeclaration of variable-name	• The variable is defined with different types each time. ⇒ Always use the same type when declaring a variable twice.
typedef initialized	• An initialize expression is described in the variable declared with typedef. ⇒ Delete the initialize expression.
uncomplete array pointer operation	• An incomplete multidimensional array has been accessed to pointer. ⇒ Specify the size of the multidimensional array.
undefined label "label" used	• The jump-address label for goto is not defined in the function. ⇒ Define the jump-address label in the function.

Table F.18 ccom100Error message(12/12)

Error message	Description and countermeasure
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.
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'spointer, ignored (NC30 only)	• 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 c ccom100 Warning Messages

Table F.19 to Table F.27 list the ccom100 compiler warning messages and their countermeasures.

Table F.19 ccom100 Warning Messages (1/9)

Warning message	Description and countermeasure
#pragma pragma-name & HANDLER both specified	• Both #pragma pragma-name and #pragma HANDLER are specified in one function. ⇒ Specify #pragma pragma-name and #pragma HANDLER exclusive to each other.
#pragma pragma-name & INTERRUPT both specified	• Both #pragma pragma-name and #pragma INTERRUPT are specified in one function. ⇒ Specify #pragma pragma-name and #pragma INTERRUPT exclusive to each other.
#pragma pragma-name & TASK both specified	• Both #pragma pragma-name and #pragma TASK are specified in one function. ⇒ Specify #pragma pragma-name and #pragma TASK exclusive to each other.
#pragma pragma-name format error	• The #pragma pragma-name is erroneously written. Processing will be continued. ⇒ Write it correctly.
#pragma pragma-name format error, ignored	• The #pragma pragma-name is erroneously written. This line will be ignored. ⇒ Write it correctly.
#pragma pragma-name not function, ignored	• A name is written in the #pragma pragma-name that is not a function. ⇒ Write it with a function name.
#pragma pragma-name's function must be predeclared, ignored	• A function specified in the #pragma pragma-name is not declared. ⇒ For functions specified in a #pragma pragmaname, write prototype declaration in advance.
#pragma pragma-name's function must be prototyped, ignored	• A function specified in the #pragma pragma-name is not prototype declared. ⇒ For functions specified in a #pragma pragmaname, write prototype declaration in advance.
#pragma pragma-name's function return type invalid,ignored	• The type of return value for a function specified in the #pragma pragma-name is invalid. ⇒ Make sure the type of return value is any type other than struct, union, or double.
#pragma pragma-name unknown switch,ignored	• The switch specified in the #pragma pragma-name is invalid. ⇒ Write it correctly.
#pragma pragma-name variable initialized, initialization ignored	• The variable specified in #pragma pragma-name is initialized. The specification of #pragma pragma-name will be nullified. ⇒ Delete either #pragma pragma-name or the initialize expression.
#pragma ASM line too long, then cut	• The line in which #pragma ASM is written exceeds the allowable number of characters = 1,024 bytes. ⇒ Write it within 1,024 bytes.

Table F.20 ccom100 Warning Messages (2/9)

Warning message	Description and countermeasure
#pragma directive conflict	• #pragma of different functions is specified for one function. ⇒ Write it correctly.
#pragma DMAC duplicate	• The same #pragma DMAC is defined twice. ⇒ Write it correctly.
#pragma DMAC variable must be far pointer for variable-name, ignored	• Variable declared by #pragma DMAC needs to be a far pointer. DMAC declaration is ignored. ⇒ Write it correctly.
#pragma DMAC variable must be unsigned int for variable-name, ignored	• Variable declared by #pragma DMAC needs to be unsigned int type. DMAC declaration is ignored. ⇒ Write it correctly.
#pragma DMAC's variable must be pre-declared, ignored	• Variable declared by #pragma DMAC needs a type declaration. ⇒ Write it correctly.
#pragma DMAC, register conflict	• Multiple variables are allocated to the same register. ⇒ Write it correctly.
#pragma DMAC, unknown register name used	• Unknown register is used in #pragma DMAC declaration. ⇒ Write it correctly.
#pragma JSRA illegal location, ignored	• Do not put #pragma JSRA inside function scope. ⇒ Write #pragma JSRA outside a function.
#pragma JSRW illegal location, ignored	• Do not put #pragma JSRW inside function scope. ⇒ Write #pragma JSRA outside a function.
#pragma PARAMETER function's address used	• The address of function specified #pragma PARAMETER is assigned to the pointer variable. ⇒ As don't assign, write correctly.
#pragma control for function duplicate, ignored	• Two or more of INTERRUPT, TASK, HANDLER, CYCHANDLER, or ALMHANDLER are specified for the same function in #pragma. ⇒ Be sure to specify only one of INTERRUPT, T A S K , H A N D L E R , C Y C H A N D L E R , o r ALMHANDLER.
#pragma unknown switch, ignored	• Invalid switch is specified to #pragma.#pragma declaration is ignored. ⇒ Write switch correctly.
'auto' is illegal storage class	• An incorrect storage class is used. ⇒ Specify the correct storage class.
'register' is illegal storage class	• An incorrect storage class is used. ⇒ Specify the correct storage class.
argument is define by 'typedef', 'typedef' ignored	• Specifier typedef is used in argument declaration. Specifier typedef will be ignored. ⇒ Delete typedef.
assign far pointer to near pointer, bank value ignored	• The bank address will be nullified when substituting the far pointer for the near pointer. ⇒ Check the data types, near or far.
assignment from const pointer to non-const pointer	• The const property is lost by assignment from const pointer to non-const pointer. ⇒ Check the statement description. If the description is correct, ignore this warning.

Table F.21 ccom100 Warning Messages (3/9)

Warning message	Description and countermeasure
assignment from volatile pointer to non-volatile pointer	• The volatile property is lost by assignment from volatile pointer to non-volatile pointer. ⇒ Check the statement description. If the description is correct, ignore this warning.
assignment in comparison statement	• You put an assignment expression in a comparison statement. ⇒ You may confuse "==" with '='. Check on it.
block level extern variable initialize forbid,ignored	• An initializer is written in extern variable declaration in a function. ⇒ Delete the initializer or change the storage class.
can't get address from register storage class variable	• The & operator is written for a variable of the storage class register. ⇒ Do not use the & operator to describe a variable of the storage class register.
can't get size of bitfield	• The bit-field is used for the operand of the sizeof operator. ⇒ Write the operand correctly.
can't get size of function	• A function name is used for the operand of the sizeof operator. ⇒ Write the operand correctly.
can't get size of function, unit size 1 assumed	• The pointer to the function is incremented (++) or decremented (--). Processing will be continued by assuming the increment or decrement value is 1. ⇒ Do not increment (++) or decrement (--) the pointer to a function.
char array initialized by wchar_t string	• The array of type char is initialized with type wchar_t . ⇒ Make sure that the types of initializer are matched.
case value is out of range	• The value of case exceeds the switch parameter range. ⇒ Specify correctly.
character buffer overflow	• The size of the string exceeded 512 characters. ⇒ Do not use more than 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 function has argument	• The function specified by #pragma CYCHANDLER or ALMHANDLER is using an argument. ⇒ The function cannot use an argument. Delete the argument.
enumerator value overflow size of unsigned char	• The enumerator value exceeded 255. ⇒ Do not use more than 255 for the enumerator; otherwise, do not specify the startup function - fchar_enumerator.
enumerator value overflow size of unsigned int	• The enumerator value exceeded 65535. ⇒ Do not use more than 65535 to describe the enumerator.
enum's bitfield	• An enumeration is used as a bit field member. ⇒ Use a different type of member.
external variable initialized,change to public	• An initialization expression is specified for an extern-declared variable. extern will be ignored. ⇒ Delete extern.

Table F.22 ccom100 Warning Messages (4/9)

Warning message	Description and countermeasure
far pointer (implicitly) casted by near pointer	• The far pointer was converted into the near pointer. ⇒ Check the data types, near or far.
function must be far	• The function is declared with the near type. ⇒ Write it correctly.
function function-name has no-used argument (variable-name)	• The variable declared in the argument to the function is not used. ⇒ Check the variables used.
handler function called	• The function specified by #pragma HANDLER is called. ⇒ Be careful not to call a handler.
handler function can't return value	• The function specified by #pragma HANDLER is using a returned value. ⇒ The function specified by #pragma HANDLER cannot use a returned value. Delete the return value.
handler function has argument	• The function specified by #pragma HANDLER is using an argument. ⇒ The function specified by #pragma HANDLER cannot use an argument. Delete the argument.
hex character is out of range	• The hex character in a character constant is excessively long. Also, some character that is not a hex representation is included after \. ⇒ Reduce the length of the hex character.
identifier (member-name) is duplicated, this declare ignored	• The member name is defined twice or more. This declaration will be ignored. ⇒ Make sure that member names are declared only once.
identifier (variable-name) is duplicated	• The variable name is defined twice or more. This declaration will be ignored. ⇒ Make sure that variable names are declared only once.
identifier (variable-name) is shadowed	• The auto variable which is the same as the name declared as an argument is used. ⇒ Use any name not in use for arguments.
illegal storage class for argument, 'extern' ignore	• An invalid storage class is used in the argument list of function definition. ⇒ Specify the correct storage class.
incomplete array access	• An incomplete multidimensional array has been accessed. ⇒ Specify the size of the multidimensional array.
incompatible pointer types	• The object type pointed to by the pointer is incorrect. ⇒ Check the pointer type.
incomplete return type	• An attempt is made to reference an return variable of incomplete type. ⇒ Check return variable.
incomplete struct member	• An attempt is made to reference an struct member of incomplete . ⇒ Define complete structs or unions first.
init elements overflow,ignored	• The initialization expression exceeded the size of the variable to be initialized. ⇒ Make sure that the number of initialize expressions does not exceed the size of the variables to be initialized.
inline function is called as normal function before, change to static function	• The function declared in storage class inline is called as an ordinary function. ⇒ Always be sure to define an inline function before using it.

Table F.23 ccom100 Warning Messages (5/9)

Warning message	Description and countermeasure
integer constant is out of range	• The value of the integer constant exceeded the value that can be expressed by unsigned long. ⇒ Use a value that can be expressed by unsigned long to describe the constant.
interrupt function called	• The function specified by #pragma INTERRUPT is called. ⇒ Be careful not to call an interrupt handling function.
interrupt function can't return value	• The interrupt handling function specified by #pragma INTERRUPT is using a return value. ⇒ Return values cannot be used in an interrupt function. Delete the return value.
interrupt function has argument	• The interrupt handling function specified by #pragma INTERRUPT is using an argument. ⇒ Arguments cannot be used in an interrupt function. Delete the argument.
invalid #pragma EQU	• The description of #pragma EQU contains an error. This line will be ignored. ⇒ Write the description correctly.
invalid #pragma SECTION, unknown section base name	• The section name in #pragma SECTION contains an error. The section names that can be specified are data, bss, program, rom, interrupt, and bas. This line will be ignored. ⇒ Write the description correctly.
invalid #pragma operand, ignored	• An operand of #pragma contains an error. This line will be ignored. ⇒ Write the description correctly.
invalid function argument	⇒ The function argument is not correctly written. • Write the function argument correctly.
invalid return type	• The expression of the return statement does not match the type of the function. ⇒ Make sure that the return value is matched to the type of the function or that the type of the function is matched to the return value.
invalid storage class for function, change to extern	• An invalid storage class is used in function declaration. It will be handled as extern when processed. ⇒ Change the storage class to extern.
Kanji in #pragma ADDRESS	• The line of #pragma ADDRESS contains kanji code. This line will be ignored. ⇒ Do not use kanji code in this declaration.
Kanji in #pragma BITADDRESS	• The line of #pragma BITADDRESS contains kanji code. This line will be ignored. ⇒ Do not use kanji code in this declaration.
keyword (keyword) are reserved for future	• A reversed keyword is used. ⇒ Change it to a different name.
large type was implicitly cast to small type	• The upper bytes (word) of the value may be lost by assignment from large type to a smaller type. ⇒ Check the type. If the description is correct, ignore this warning.
mismatch prototyped parameter type	• The argument type is not the type declared in prototype declaration. ⇒ Check the argument type.

Table F.24 ccom100 Warning Messages (6/9)

Warning message	Description and countermeasure
meaningless statements deleted in optimize phase	• Meaningless statements were deleted during optimization. ⇒ Delete meaningless statements.
meaningless statement	• The tail of a statement is "==". ⇒ You may confuse "=" with '=='. Check on it.
mismatch function pointer assignment	• The address of a function having a register argument is substituted for a pointer to a function that does not have a register argument (i.e., a nonprototyped function). ⇒ Change the declaration of a pointer variable for function to a prototype declaration.
multi-character character constant	• A character constant consisting of two characters or more is used. ⇒ Use a wide character (L'xx') when two or more characters are required.
near/far is conflict beyond over typedef	• The type defined by specifying near/far is again defined by specifying near/far when referencing it. ⇒ Write the type specifier correctly.
No hex digit	• The hex constant contains some character that cannot be used in hex notation. ⇒ Use numerals 0 to 9 and alphabets A to F and a to f to describe hex constants.
No initialized of variable-name	• It is probable that the register variables are used without being initialized. ⇒ Make sure the register variables are assigned the appropriate value.
No storage class & data type in declare, global storage class & int type assumed	• The variable is declared without storage-class and type specifiers. It will be handled as int when processed. ⇒ Write the storage-class and type specifiers.
non-initialized variable “variable-name” is used	• It is probable that uninitialized variables are being referenced. ⇒ Check the statement description. This warning can occur in the last line of the function. In such a case, check the description of the auto variables, etc. in the function. If the description is correct, ignore this warning.
non-prototyped function used	• A function is called that is not declared of the prototype. This message is output only when you specified the Wnon_prototype option. ⇒ Write prototype declaration. Or delete the option “- Wnon_prototype”.
non-prototyped function declared	• A prototype declaration for the defined function cannot be found. (Displayed only when the - 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.

Table F.25 com100 Warning Messages (7/9)

Warning message	Description and countermeasure
old style function declaration	• The function definition is written in format prior to ANSI (ISO) C. ⇒ Write the function definition in ANSI (ISO) format.
prototype function is defined as non-prototype function before.	• The non-prototyped function is redefine prototype-declaration. ⇒ Unite ways to declare function type.
redefined type	• Redwfine typedef. ⇒ Check typedef.
redefined type name of (qualify)	• The same identifier is defined twice or more in typedef. ⇒ Write identifier correctly.
register parameter function used before as stack parameter function	• The function for register argument is used as a function for stack argument before. ⇒ Write a prototype declaration before using the function.
RESTRICT qualifier can set only pointer type.	• The RESTRICT qualifier is declared outside a pointer. ⇒ Declare it in only a pointer.
section name 'interrupt' no more used	• The section name specified by "pragma SECTION uses 'interrupt'. ⇒ A section name 'interrupt' cannot be used. Change it to another.
size of incomplete type	• An undefined structure or union is used in the operand of the size of operator. ⇒ Define the structure or union first.
size of incomplete array type	• The number of elements of an array defined as an operand of the size of operator is unknown. ⇒ Define the structure or union first.
	• An attempt is made to find size of of an array of unknown size. This is an invalid size. ⇒ Specify the size of the array.
size of void	• An attempt is made to find the size of void. This is an invalid size. ⇒ The size of void cannot be found.
standard library ”function-name()” need “include-file name”	• This standard library function is used without its header file included. ⇒ Be sure to include the header file.
static variable in inline function	• static data is declared within a function that is declared in storage class inline. ⇒ Do not declare static data in an inline function.
string size bigger than array size	• The size of the initialize expression is greater than that of the variable to be initialized. ⇒ Make sure that the size of the initialize expression is equal to or smaller than the variable.
string terminator not added	• Since the variable to be initialized and the size of the initialize expression are equal, '\0' cannot be affixed to the character string. ⇒ Increase a element number of array.
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.

Table F.26 ccom100 Warning Messages (8/9)

Warning message	Description and countermeasure
task function called	• The function specified by #pragma TASK is called. ⇒ Be careful not to call a task function.
task function can't return value	• The function specified by #pragma TASK is using a return value. ⇒ The function specified by #pragma TASK cannot use return values. Delete the return value.
task function has invalid argument	• The function specified with #pragma TASK uses arguments. ⇒ Any function specified with #pragma TASK cannot use arguments. Delete the arguments.
this comparison is always false	• Comparison is made that always results in false. ⇒ Check the conditional expression.
this comparison is always true	• Comparison is made that always results in true. ⇒ Check the conditional expression.
this feature not supported now, ignored	• This is a syntax error. Do not this syntax because t is reserved for extended use in the future. ⇒ Write the description correctly.
this function used before with non-default argument	• A function once used is declared as a function hat has a default argument. ⇒ Declare the default argument before using a unction.
this interrupt function is called as normal function before	• A function once used is declared in #pragma NTERRUPT. ⇒ An interrupt function cannot be called. Check the ontent of #pragma.
too big octal character	• The character constant or the octal constant in he character string exceeded the limit value (255 n decimal). ⇒ Do not use a value greater than 255 to describe he constant.
too few parameters	• Arguments are insufficient compared to the number f arguments declared in prototype declaration. ⇒ Check the number of arguments.
too many parameters	• Arguments are excessive compared to the number f arguments declared in prototype declaration. ⇒ Check the number of arguments.
unknown #pragma STRUCT xxx	• #pragma STRUCTxxx cannot be processed. his line will be ignored. ⇒ Write correctly.
Unknown debug option (-dx)	• The option -dx cannot be specified. ⇒ Specify the option correctly.
Unknown function option (-Wxxx)	• The option -Wxxx cannot be specified. ⇒ Specify the option correctly.
Unknown function option (-fx)	• The option -fx cannot be specified. ⇒ Specify the option correctly.
Unknown function option (-gx)	• The option -gx cannot be specified. ⇒ Specify the option correctly.
Unknown optimize option (-mx)	• The option -mx cannot be specified. ⇒ Specify the option correctly.
Unknown optimize option (-Ox)	• The option -Ox cannot be specified. ⇒ Specify the option correctly.
Unknown option (-x)	• The option -x cannot be specified. ⇒ Specify the option correctly.

Table F.27 ccom100 Warning Messages (9/9)

Warning message	Description and countermeasure
unknown pragma pragma-specification used	• Unsupported #pragma is written. ⇒ Check the content of #pragma. *This warning is displayed only when the Wunknown_pragma (-WUP) option is specified.
wchar_t array initialized by char string	• The initialize expression of the wchar_t type is nitialized by a character string of the char type. ⇒ Make sure that the types of the initialize expression re matched.
zero divide in constant folding	• The divisor in the divide operator or remainder alculation operator is 0. ⇒ Use any value other than 0 for the divisor.
zero divide, ignored	• The divisor in the divide operator or remainder alculation operator is 0. ⇒ Use any value other than 0 for the divisor.
zero width for bitfield	• The bit-field width is 0. ⇒ Write a bit-field equal to or greater than 1.
no const in previous declaretion	• The function or variable declaration without const qualification is const-qualified on the entity definition side. ⇒ Make sure the function or variable declaration and the const qualification on the entity definition side are matched.

Appendix G Using gensni or the stack information File Creation Tool for Call Walker

Before Call Walker or the stack analysis tool of the High-performance Embedded Workshop can be used, you must have stack information files (extension .sni) as the input files for it.
You use gensni or the stack information file creation tool for Call Walker to create these stack information files from the absolute module file.

G.1 Starting Call Walker

To start Call Walker, select "Call Walker" that is registered to the High-performance Embedded Workshop or select the tool from the Tools menu of the High-performance Embedded Workshop.
After starting Call Walker, choose Import Stack File from the File menu and select a stack information file as the input file for Call Walker.

G.2 Outline of gensni

G.2.1 Processing Outline of gensni

gensni is the tool to create .sni files for Call Walker.
gensni generates a stack information file (extension .sni) by processing the absolute module file (extension .x30). Before gensni can be used, there must be an absolute module file (extension .x30) available. Specify the compile option "-finfo" during compilation to generate that file.
The processing flow of NC100 is shown in Figure G.1

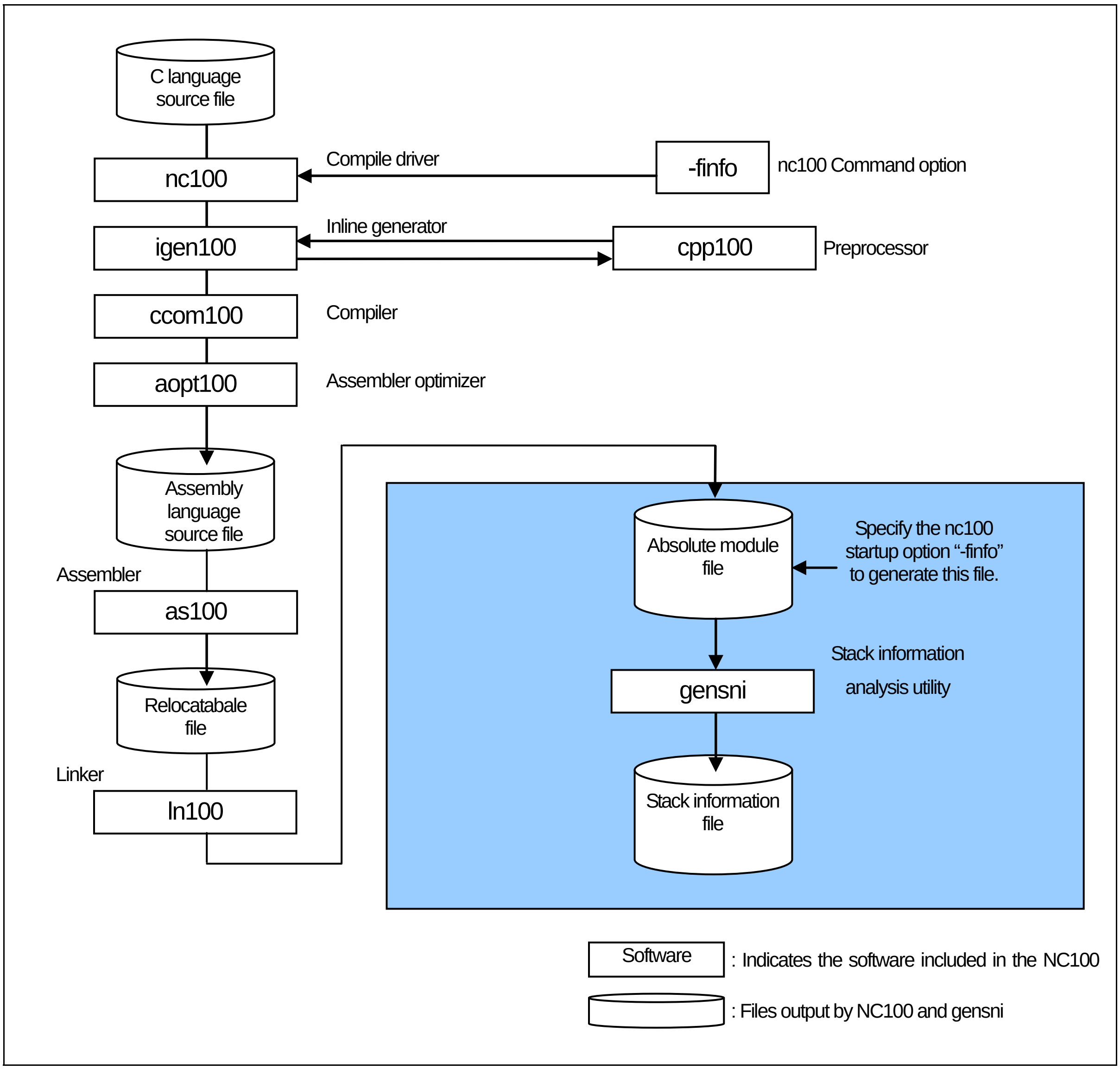


Figure G.1 Processing flow of NC100

G.3 Starting gensni

If Call Walker is started from the High-performance Embedded Workshop, gensni is automatically executed. However, if Call Walker is started from other than the High-performance Embedded Workshop, gensni is not automatically executed. In this case, start gensni from the Windows command prompt.

G.3.1 Input format

To start gensni, specify an input file name and startup option according to the input format shown below.

```
% gensni△[Command option]△Absolute module file(extension.x30)

% : Denotes the prompt
< > : Denotes the essential items.
[ ] : Denotes the items that need to be written when necessary.
△ : Denotes a space.
When writing multiple startup options, separate each with a space.
```

Figure G.2 gensni command input format

To use gensni, specify both of the following in the startup options of this compiler

- Inspector information output -finfo option
- Debug information output -g option

to generate absolute module files (extension ".x30").

An input example is shown below. In the input example here, the following option is specified in gensni.

- Information output to a specified file -o option

(By default, the information is output to a file named after the input file by changing the file extension from ".x30" to ".sni."

```
Generate an absolute module file :

%nc100 –g –finfo ncrt0.a30 sample.c<RET>
R32C/100 Series C Compiler V.X.XX Release XX
Copyright(C) XXXX(XXXX-XXXX). Renesas Technology Corp.
and Renesas Solutions Corp., All rights reserved.
ncrt0.a30
sample.c

%

Generate stack information file:

%gensni -o sample ncrt0.x30<RET>

sample.sni is created.

%

<RET> : The input of the return key is shown.
```

Figure G.3 gensni command input example

G.3.2 Option References

The startup options of gensni are listed in Table G.1.

Table G.1 gensni Command option

Option	short form	function
-o file name	None	Specify a stack information file name. ● If this option is not specified, the stack information file is named after the input file by changing its file extension to ".sni." ● If an extension is specified .sni file name, the specified extension is changed to ".sni." If no extensions are specified, the extension ".sni" is assumed.
-V	None	Shows the startup message of gensni and terminates processing without performing anything. No stack information files are generated.

-o Stack information file

Specify a stack information file name

Function:
- If this option is not specified, the stack information file is named after the input file by changing its file extension to ".sni."
- If an extension is specified the stack information file name, the specified extension is changed to ".sni." If no extensions are specified, the extension ".sni" is assumed.

Description: Use of this option permits you to change the stack information file name as necessary. The extension can also be changed.

-V

Terminate processing after showing the startup message of gensni

Function: Shows the startup message of gensni and terminates processing without performing anything.
- No stack information files are generated.

G.4 Error Messages of gensni

G.4.1 Error Messages

Table G.2 lists the error messages output by gensni along with the contents of errors and the corrective actions to be taken.

Table G.2 List of Error Messages of gensni

Error Messages	Content of Error and Corrective Action
usage: gensni [-V][-o out_file] in_file	• The input format is incorrect. ⇒ Specify a correct input format.
Can't open file: XXX	• The absolute module file cannot be opened. ⇒ Check whether the file exists and the file attribute.
Can't create file: XXX	• The stack information file cannot be created. ⇒ Check the file and folder attributes. Check the free disk space available.
Illegal file format: XXX	• The content of the absolute module file is incorrect. No stack information file can be created. ⇒ Check whether the absolute module file is the one that you created with NC100. Also check whether -finfo and -g are specified in the compiler options.
Not enough memory	• Memory could not be allocated for gensni. ⇒ Check the available memory size of your PC.

R32C/100 Series C Compiler Package V.1.01
C Compiler User's Manual

Publication Date: Aug. 1, 2007 Rev.1.00

Published by: Sales Strategic Planning Div.
Renesas Technology Corp.

Edited by: Microcomputer Tool Development Department
Renesas Solutions Corp.

R32C/100 Series C Compiler Package V.1.01
C Compiler User's Manual

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan

REJ10J1657-0100