User's Manual

CC-RX

Compiler

User's Manual

Applicable Revision

V2.00.00 to V3.06.00

Target Device

RX Family

Target CPU Cores:

RXv1, RXv2, RXv3

Renesas Electronics
www.renesas.com

Rev.1.13 2023.12

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION ("Vulnerability Issues"). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

How to Use This Manual

This manual describes the role of the CC-RX compiler for developing applications and systems for RX family, and provides an outline of its features.

Readers	This manual is intended for users who wish to understand the functions of the CC-RX and design software and hardware application systems.
Purpose	This manual is intended to give users an understanding of the functions of the CC-RX to use for reference in developing the hardware or software of systems using these devices.
Organization	This manual can be broadly divided into the following units. 1.GENERAL 2.COMMAND REFERENCE 3.OUTPUT FILES 4.COMPILER LANGUAGE SPECIFICATIONS 5.ASSEMBLY LANGUAGE SPECIFICATIONS 6.SECTION SPECIFICATIONS 7.LIBRARY FUNCTIONAL SPECIFICATION 8.STARTUP 9.FUNCTION CALL INTERFACE SPECIFICATIONS 10.MESSAGES 11.Usage Notes A.QUICK GUIDE
How to Read This Manual	It is assumed that the readers of this manual have general knowledge of electricity, logic circuits, and microcontrollers.

Conventions

Data significance:	Higher digits on the left and lower digits on the right
Active low representation:	XXX (overscore over pin or signal name)
Note:	Footnote for item marked with Note in the text
Caution:	Information requiring particular attention
Remarks:	Supplementary information
Numeric representation:	Decimal ... XXXX Hexadecimal ... 0xXXXX

TABLE OF CONTENTS

1. GENERAL

This document is the user's manual for the RX family C/C++ compiler CC-RX V2.00 to V3.06.
This chapter provides an overview of the processing of CC-RX and provides an example of program development.

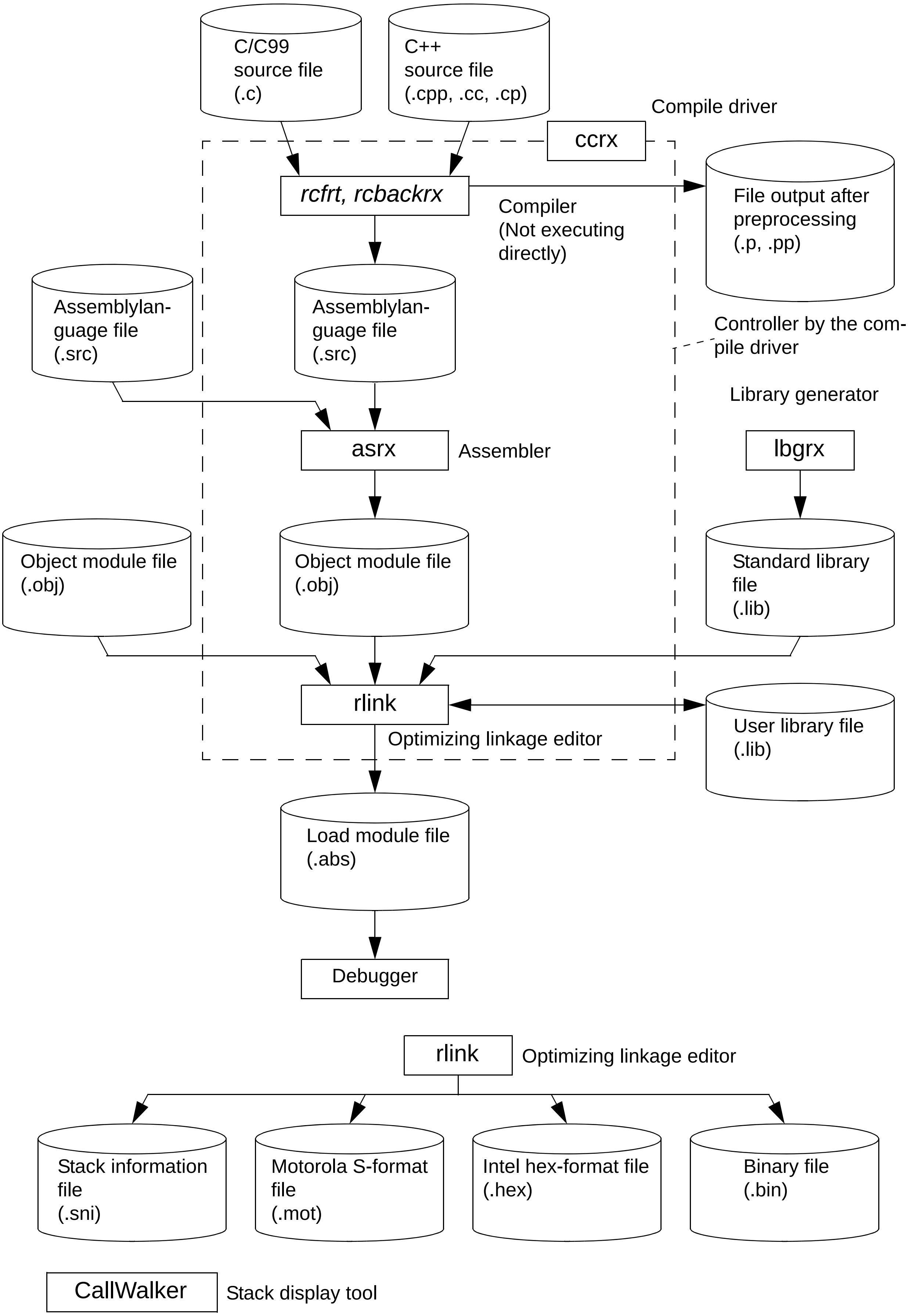
1.1 Overview

CC-RX is a program that converts programs written in C/C++ or assembly language to a machine language.
CC-RX is comprised of the four executable files listed below.

(1) ccrx: Compile driver
(2) asrx: Assembler Optimizer
(3) rlink: Optimizing linkage editor
(4) lbgrx: Library generator

Figure 1.1 illustrates the CC-RX processing flow.

Figure 1.1 CC-RX Processing Flow



1.2 Copyrights

This software uses LLVM and Protocol Buffers.

- LLVM is copyright of University of Illinois at Urbana-Champaign.
- Protocol Buffers is copyright of Google Inc.

Other software components are copyright of Renesas Electronics Corporation.

1.3 Special Features

The RX family C/C++ compiler package (CC-RX) is equipped with the following special features.

(1) Language specifications in accordance with ANSI standard
The C, C99, and C++ language specifications conform to the ANSI standard. Coexistence with prior C language specifications (K&R specifications) is also provided.

(2) Advanced optimization
Code size and speed priority optimization for the C compiler are offered.

(3) Improvement to description ability
C language programming description ability has been improved due to enhanced language specifications.

(4) High portability
The single CC-RX supports all microcontrollers. This makes it possible to use a uniform language specification, and facilitates porting between microcontrollers.
In addition, the industry-standard DWARF2/3 format is used for debugging information.

1.4 Limits

1.4.1 Limits of Compiler

Table 1.1 shows the translation limits of the compiler.
Source programs must be created to fall within these translation limits.

Table 1.1 Translation Limits of Compiler

No.	Classification	Item	Translation Limit
1	Startup	Total number of macro names that can be specified using the **define** option	Unlimited
2		Number of characters in a file name	Unlimited (depends on the OS)
3	Source program	Number of characters in one line	32768
4		Number of source program lines in one file	Unlimited
5		Total number of source program lines that can be compiled	Unlimited
6	Preprocessing	Nesting levels of files in an **#include** statement	Unlimited
7		Total number of macro names in a **#define** statement	Unlimited
8		Number of parameters that can be specified using a macro definition or macro call operation	Unlimited
9		Number of expansions of a macro name	Unlimited
10		Nesting levels of conditional inclusion	Unlimited
11		Total number of operators and operands that can be specified in an **#if** or **#elif** statement	Unlimited

No.	Classification	Item	Translation Limit
12	Declaration	Number of function definitions	Unlimited
13		Number of external identifiers used for external linkage	Unlimited
14		Number of valid internal identifiers used in one function	Unlimited
15		Number of pointers, arrays, and function declarators that qualify the basic type	16
16		Number of array dimensions	6
17		Size of arrays and structures	2147483647 bytes
18	Statement	Nesting levels of compound statements	Unlimited
19		Nesting levels of statements in a combination of repeat (**while**, **do**, and **for**) and select (**if** and **switch**) statements	4096
20		Number of compound statements that can be written in one function	2048
21		Number of **goto** labels that can be specified in one function	2147483646
22		Number of **switch** statements	2048
23		Nesting levels of **switch** statements	2048
24		Number of **case** labels that can be specified in one **switch** statement	2147483646
25		Nesting levels of **for** statements	2048
26	Expression	Number of characters in a string	32766
27		Number of parameters that can be specified using a function definition or function call operation	2147483646
28		Total number of operators and operands that can be specified in one expression	About 500
29	Standard library	Number of files that can be opened simultaneously in an **open** function	Variable*1
30	Section	Length of section name*2	8146
31		Number of sections that can be specified in **#pragma section** in one file	2045
32		Maximum size of each section	4294967295 bytes
33	Output files	Maximum number of characters per line of assembly source code that can be output	8190

Notes 1. For details, refer to section 8.3.2 Initial Setting.

Notes 2. Since the assembler's limit of number of characters in one line is applied to the length of a section name when generating an object, the length that can be specified in #pragma section or the section option is shorter than this limit.

1.4.2 Limits of Assembler

Table 1.2 shows the translation limits of the assembler.
Source programs must be created to fall within these translation limits.

Table 1.2 Translation Limits of Assembler

No.	Item	Translation Limit
1	Number of characters in one line	32760
2	Symbol length	Number of characters in one line*1
3	Number of symbols	Unlimited
4	Number of externally referenced symbols	Unlimited
5	Number of externally defined symbols	Unlimited
6	Maximum size for a section	0FFFFFFFFH bytes
7	Number of sections	65265 (with debugging information) or 65274 (without debugging information)
8	File include	Nesting levels of 30
9	String length	Number of characters in one line*1
10	Number of characters in a file name	Number of characters in one line*1
11	Number of characters in an environment variable setting	2048 bytes
12	Number of macro definitions	65535
13	Number of characters in a subcommand file	32767*2
14	Number of characters in one line of a subcommand file	2048

Notes 1. The limit may become a smaller value depending on the string length specified in the same line.

Notes 2. The limit may become a smaller value because this value includes the number of characters for other assembler options.

1.5 License

A license manager manages licenses to the compilers.

If you have a license, the compiler will operate as the Standard or Professional edition depending on the license you are using.

Refer to section 1.6, Standard and Professional Editions, for more on the Standard and Professional editions.

If the license manager is not able to recognize a Standard or Professional license, the compiler operates as the free evaluation edition.

Refer to section 1.7, Free Evaluation Editions, for more on the free evaluation edition.

For details of the licenses and the license manager, refer to the User's Manual of the License Manager.

Use V2.00 or later versions of the license manager for V2.06 and later versions of CC-RX.

1.6 Standard and Professional Editions

There are two editions of the compilers, the Standard and the Professional editions.

The Standard editions support an ANSI-compliant C-language specification, and also provide the essential features for writing programs for embedded systems.

As well as the features of the Standard editions, the Professional editions have additional features which help to improve the quality of the customer's programs and shorten development periods.

The additional features of Professional editions are available through compiler options, #pragma directives and libraries.

For descriptions of the options only available for the Professional editions, refer to section 2.5, Options, or the descriptions of the individual options.

For descriptions of the #pragma directives that only the Professional editions support, refer to section 4.2.3, #pragma Directive.

1.7 Free Evaluation Editions

The free evaluation editions have a trial period of 60 days from the day of the first building by the compiler over which you can use features equivalent to those of the Professional editions.

After that period, the additional features of the Professional editions are no longer available, and a restriction becomes applicable to the sizes produced by linkage.

- The restriction on the section sizes which can be allocated to the ROM area is up to 128 Kbytes in total. A linker error occurs when the size exceeds 128 Kbytes.

The version number of the optimizing linkage editor is prefixed by W while a compiler is operating as an evaluation edition and by V when it is operating as a commercial edition.

Examples are given below.

- Version of a free evaluation edition:
  Renesas Optimizing Linker W1.01.01 [25 Apr 2014]
- Version of a commercial edition:
  Renesas Optimizing Linker V1.01.01 [25 Apr 2014]

We do not supply the following services for the evaluation editions.Consider purchasing a commercial edition if you require them.

- Technical support
- E-mail delivery of items such as information on revisions

2. COMMAND REFERENCE

This appendix describes the detailed specifications of each command included in the build tool.

2.1 Overview

The RX family C/C++ compiler generates a file executable in the target system from the source program written in C language, C99 language, C++ language, or assembly language.

In this compiler, a single driver controls multiple phases from preprocessing to linkage.

The following describes processing in each phase.

(1) Compiler
This processes preprocessing directives, comments, and optimization for the C source program and generates an assembly-language source file.

(2) Preprocessor
This processes the preprocessing directives in the source program.
Only when the -output=prep option is specified, it outputs the preprocessed file.

(3) Parsing section
This parses the C source program and then converts it to the internal data representation for the compiler.

(4) Optimizing section
This optimizes the internal data representation converted from the C source program.

(5) Code generating section
This converts the internal data representation to an assembly-language source program.

(6) Assembler
This converts the assembly-language source program to machine-language instructions and generates a relocatable object module file.

(7) Optimizing linkage editor
This links object module files, link directive files, and library files and generates an object file (load module file) executable in the target system.

2.2 Input/Output Files

The following shows the files input to and output from the RX family C/C++ compiler.

Table 2.1 Input/Output Files for the RX Family C/C++ Compiler

File Type	Extension	I/O	Description
C source program file	.c	Input	A source file written in C99 language. This file is created by the user.
C++ source program file	.cpp, .cp, and .cc	Input	A source file written in C++ language. This file is created by the user.
Include file	Optional	Input	A file referenced by the source file and written in C, C99, C++, or assembly language. This file is created by the user.
Preprocessor expansion file for the C program	.p	Output	A file output as a result of preprocessing applied to an input C-language or the C99-language source program. An ASCII image file. This is output when the -output=prep option is specified.
Preprocessor expansion file for the C++ program	.pp	Output	A file output as a result of preprocessing applied to an input C++-language source program. An ASCII image file. This is output when the -output=prep option is specified.

File Type	Extension	I/O	Description
Assembly-source program file	.src	Output	An assembly-language file generated from a C, C99, or C++ source file through compilation.
	.src	Input	A source file written in assembly language.
List file for the assembly program	.lst	Output	A list file containing the assembly result information. This is output when the -listfile option is specified. The output contents are selected with the -show option.
Relocatable object program file	.obj	Output	An ELF-format file that contains the machine-language information, the relocation information about the allocation addresses of machine-language instructions, and symbol information.
Absolute load module file	.abs	Output	An ELF-format file for the object code generated as a result of linkage. This is an input file when a hex file is output.
Linkage list file	.map	Output	A list file containing the linkage result information. This is output when the -list option is specified. The output contents are selected with the -show option.
Library file	.lib	Output	A file where multiple object module files are registered.
Library list file	.lbp	Output	A list file containing the result information of generation of the library. This is output when the -list option is specified. The output contents are selected with the -show option.
Library backup file	.lbk	Output	File type for saving the contents of original library files before they are overwritten by the library generator.
Hex file (Motorola S-format file)	.mot	Output	A Motorola S-format file in the hex format converted from the load module file.
Hex file (Intel (expansion) hex format file)	.hex	Output	An Intel (expansion) file in the hex format converted from the load module file.
Hex file (binary format file)	.bin	Output	A binary file in the hex format converted from the load module file.
Stack information file	.sni	Output	A stack information file. This is output when the -stack option is specified.
Debugging information file	.dbg	Output	A debugging information file. This is output when the -sdebug option is specified.
Object file including a definition specified with a file having extension td	.rti	Output	An object file including a definition specified with a file having extension td.
Calling information file	.cal	Output	A calling information file. This is output by CallWalker.
External symbol assignment information file	.bls	Output	An external symbol assignment information file. This is output at linkage when the -map option is specified.
	.bls	Input	An external symbol assignment information file. This is specified as an input file for the -map option at compilation.

File Type	Extension	I/O	Description
Jump table file (assembly language)	.jmp	Output	An assembler source file for the jump table that branches the external definition symbol. This is output when the -jump_entries_for_pic option is specified.
Symbol address file (assembly language)	.fsy	Output	An assembler source file that describes the external definition symbol in an assembler directive. This is output when the -fsymbol option is specified.
C++ language function support file	.td, .ti, .pi, and .ii	Output	An information file that supports the C++ language function.
Tool usage information file	.ud .udm	Output	File which is output for collecting tool usage information

2.3 Environment Variables

Environment variables are listed below. Note that there might be the case where environment variables are set in the integrated development environment. Also refer to the user's manual of the integrated development environment.

Table 2.2 Environment Variables

No.	Environment Variable	Description	Default When Specification is Omitted
1	path	Specifies a storage directory for the execution file.	[V3.05 or earlier] Specification cannot be omitted.
2	BIN_RX	Specifies the directory in which ccrx is stored.	<ccrx storage directory> [V3.05 or earlier] Specification cannot be omitted when the lbgrx command is used.
3	ISA_RX *1	Selects an instruction-set architecture. <Instruction-set architectures> RXV1 RXV2 RXV3 [V3.00.00 or later]	No value is set when the specification is omitted.
4	INC_RX	Specifies a directory in which an include file of the compiler is stored.	<ccrx storage directory> \..\include
5	INC_RXA	Specifies a directory in which an include file of the assembler is stored.	No value is set when the specification is omitted.
6	TMP_RX	Specifies a directory in which a temporary file is generated.	OS settings
7	HLNK_LIBRARY1 HLNK_LIBRARY2 HLNK_LIBRARY3	Specifies a default library name for the optimizing linkage editor. Libraries which are specified by a library option are linked first. Then, if there is an unresolved symbol, the default libraries are searched in the order of 1, 2, 3.	No value is set when the specification is omitted.

No.	Environment Variable	Description	Default When Specification is Omitted
8	HLNK_TMP	Specifies a folder in which the optimizing linkage editor generates temporary files. If HLNK_TMP is not specified, the temporary files are created in the current folder.	No value is set when the specification is omitted.
9	HLNK_DIR	Specifies an input file storage folder for the optimizing linkage editor. The search order for files which are specified by the input or library option is the current folder, then the folder specified by HLNK_DIR. However, when a wild card is used in the file specification, only the current folder is searched.	No value is set when the specification is omitted.
10	CPU_RX *1	Specifies the CPU type. <CPU types> RX600 RX200	No value is set when the specification is omitted.

*1) When both ISA_RX and CPU_RX are defined, ISA_RX takes precedence.

2.4 Operating Instructions

This section describes how to operate the RX family C/C++ compiler.

The commands will take options from left to right on the command-line. When two or more options with conflicted meanings are selected and it will take neither error nor warning, the right-side option will be enabled. This results are different according to each options. For more details, please confirm each options' descriptions.

(1) Operating Tools

(a) Compiler (ccrx)
ccrx is the startup command for the compile driver.
Compilation, assemble, and linkage can be performed using this command.
When the extension of an input file is ".s", ".src", ".S", or ".SRC", the compiler interprets the file as an assembly-language file (.src, .s) and initiates the assembler.
A file with an extension other than those above is compiled as a C/C++ source file (.c, .cpp).

Two or more input files can be specified at the same time. Cases where two or more C/C++ language source files are specified as input files at the same time are referred to as "batch compilation."

[Command description format]

```
ccrx [Δ<option> …][Δ<file name>[ Δ<option> …] …]
       <option>: -<option>[=<suboption>[=<suboption>]][, …]
```

[ ] : Can be omitted
... : Pattern in proceeding [ ] can be repeated
{ } : Select from items delimited by the pipe symbol ("|")
Δ : One or more spaces

(b) Assembler (asrx)
asrx is the startup command for the assembler.
[Command description format]

```
asrx [Δ<option> …][ Δ<file name>[ Δ<option> …] …]
       <option>: -<option>[=<suboption>][, …]
```

(c) Optimizing Linkage Editor (rlink)
rlink is the startup command for the optimizing linkage editor.
The optimizing linkage editor has the following functions as well as the linkage processing.

- Optimizes relocatable files at linkage
- Generates and edits library files
- Converts files into Motorola S type files, Intel hex type files, and binary files

[Command description format]

```
rlink [Δ<option> …][ Δ<file name>[ Δ<option> …] …]
        <option>: -<option>[=<suboption>][, …]
```

(d) Library Generator (lbgrx)
lbgrx is the startup command for the library generator.
[Command description format]

```
lbgrx [Δ<option> …]
        <option>: -<option>[=<suboption>][, …]
```

(2) Command Description Examples

(a) Compilation, Assemble, and Linkage by One Command
Perform all steps below by a single command.

- Compile C/C++ source files (tp1.c and tp2.c) in **ccrx**.
- After compilation, assemble the files in **asrx**.
- After assemble, link the files in **rlink** to generate an absolute file (tp.abs).

[Command description]

```
ccrx -isa=rxv1 -output=abs=tp.abs tp1.c tp2.c
```

Remarks 1. When the output type specification of the **output** option is changed to **-output=sty**, the file after linkage will be generated as a Motorola S type file.

Remarks 2. An intermediate file generated during the absolute file generation process (assembly-language file or relocatable file) is not saved. Only a file of the type specified by the **output** option is to be generated.

Remarks 3. In order to specify assemble options and linkage options that are valid for only the assembler and optimizing linkage editor in **ccrx**, use the **-asmcmd**, **-lnkcmd**, **-asmopt**, and **-lnkopt** options.

Remarks 4. Object files that are to be linked are allocated from address 0. The order of the sections is not guaranteed. In order to specify the allocation address or section allocation order, specify options for the optimizing linkage editor using the **-lnkcmd** and **-lnkopt** options.

(b) Compilation and Assemble by One Command
Perform all steps below by a single command, and initiate the linker with another command to generate tp.abs.

- Compile C/C++ source files (tp1.c and tp2.c) in **ccrx**.
- After compilation, assemble the files in **asrx** to generate relocatable files (tp1.obj and tp2.obj).

[Command description]

```
ccrx -isa=rxv1 -output=obj tp1.c tp2.c
rlink -form=abs -output=tp.abs -subcommand=cmd.sub tp1.obj tp2.obj
```

Remarks 1. When the **-output=obj** option is specified in **ccrx**, **ccrx** generates relocatable files.

Remarks 2. In order to change relocatable file names, their C/C++ source files have to be input in **ccrx**, one file each.

Remarks 3. When the **form** option in **rlink** is changed to **-form=sty**, the file after linkage will be generated as a Motorola S type file.

(c) Compilation, Assemble, and Linkage by Separate Commands
Individually perform each step below by a single command.

- Compile C/C++ source files (tp1.c and tp2.c) in **ccrx** to generate assembly-language files (tp1.src and tp2.src).
- Assemble the assembly-language files (tp1.src and tp2.src) in **asrx** to generate relocatable files (tp1.obj and tp2.obj).
- Link the relocatable files (tp1.obj and tp2.obj) in **rlink** to generate an absolute file (tp.abs).

[Command description]

```
ccrx -isa=rxv1 -output=src tp1.c tp2.c
asrx tp1.src tp2.src
rlink -form=abs -output=tp.abs -subcommand=cmd.sub tp1.obj tp2.obj
```

Remark When the **-output=src** option is specified in **ccrx**, **ccrx** generates assembly-language files.

(d) Assemble and Linkage by One Command
Perform all steps below by a single command.

- Assemble assembly-language files (tp1.src and tp2.src) in **asrx**.
- After assemble, link the files in **rlink** to generate an absolute file (tp.abs).

[Command description]

```
ccrx -isa=rxv1 -output=abs=tp.abs tp1.src tp2.src
```

Remark Object files that are to be linked are allocated from address 0. The order of the sections is not guaranteed. In order to specify the allocation address or section allocation order, specify options for the optimizing linkage editor using the **-lnkcmd** and **-lnkopt** options.

(e) Assemble and Linkage by Separate Commands
Individually perform each step below by a single command.

- Assemble assembly-language files (tp1.src and tp2.src) in **asrx** to generate relocatable files (tp1.obj and tp2.obj).
- Link the relocatable files (tp1.obj and tp2.obj) in **rlink** to generate an absolute file (tp.abs).

[Command description 1]

```
ccrx -isa=rxv1 -output=obj tp1.src tp2.src
rlink -form=abs -output=tp.abs -subcommand=cmd.sub tp1.obj tp2.obj
```

[Command description 2]

```
asrx -isa=rxv1 tp1.src tp2.src
rlink -form=abs -output=tp.abs -subcommand=cmd.sub tp1.obj tp2.obj
```

(f) Create a List File of Existing Libraries
Create a list of lib1.lib with the name of lib1.lbp.

[Command description]

```
rlink -form=library -list -library=lib1.lib
```

2.5 Options

This section describes the options for the RX family C/C++ compiler in each processing phase.
Compile phase: Refer to 2.5.1 Compile Options.
Assembly phase: Refer to 2.5.2 Assembler Command Options.
Link phase: Refer to 2.5.3 Optimizing Linkage Editor (rlink) Options.
Library generation phase: Refer to 2.5.4 Library Generator Options.

2.5.1 Compile Options

The types and explanations for options of the compile phase are shown below.

Classification	Option	Description
Source Options	-lang	Specifies the language to assume in compiling the source file.
	-include	Specifies the names of folders that hold include files.
	-preinclude	Specifies the names of files to be included at the head of each compiling unit.
	-define	Specifies macro definitions.
	-undefine	Specifies disabling of predefined macros.
	-message	Information-level messages are output.
	-nomessage	Specifies the numbers of information-level messages to be disabled.
	-change_message	Changes the levels of compiler output messages.
	-no_warning [V2.08.00 or later]	Disables the output of warnings and information-level messages.
	-file_inline_path	Specifies the names of folders that hold files for inter-file inline expansion.
	-comment	Selects permission for comment (/* */) nesting.
	-truncated_address_initializer [V3.01.00 or later]	Allows 1-byte or 2-byte type external variables in C source files to be initialized by address values.
	-check	Checks compatibility with an existing program.
	-misra2004 [Professional Edition only]	Checks the source code against the MISRA-C: 2004 rules.
	-misra2012 [Professional Edition only] [V2.04.00 or later]	Checks the source code against the MISRA-C: 2012 rules.
	-ignore_files_misra [Professional Edition only]	Selects files that will not be checked against the MISRAC: 2004 rules or MISRA-C: 2012 rules.
	-check_language_extension [Professional Edition only]	Enables complete checking against the MISRA-C: 2004 rules or MISRA-C: 2012 rules for parts of the code where this would otherwise be suppressed due to use of an extended specification.
	-misra_intermodule [Professional Edition only] [V3.01.00 or later]	Checks the source code in multiple files against the MISRA-C:2012 rules.

Classification	Option	Description
Object Options	-output	Selects the output file type.
	-noline	Selects the non-output of **#line** in preprocessor expansion.
	-debug	Debugging information is output to the object files.
	-nodebug	Debugging information is not output to the object files.
	-g_line [V3.02.00 or later]	Enhances source debugging information during optimization.
	-section	Changes section names to be changed.
	-stuff	Variables are allocated to sections that match their alignment values.
	-nostuff	Alignment values of variables are ignored in allocating the variables to sections.
	-instalign4	Instructions at branch destinations are aligned with 4-byte boundaries.
	-instalign8	Instructions at branch destinations are aligned with 8-byte boundaries.
	-noinstalign	Instructions at branch destinations have no specific alignment.
	-nouse_div_inst	Generates code in which no DIV, DIVU, FDIV or DDIV instructions are used for division and modular division.
	-create_unfilled_area [V2.03.00 or later]	[To be supported by V2.03 and later versions] Makes spaces created by .OFFSET unfilled.
	-stack_protector/ -stack_protector_all [Professional Edition only] [V2.04.00 or later]	This option generates a code for detection of stack smashing.
	-avoid_cross_boundary_prefetch [V2.07.00 or later]	Prevents the reading of data across 4-byte boundaries in prefetching for string manipulation instructions.
	-insert_nop_with_label [V2.08.00 or later]	This option inserts a local label and nop instruction.
	-control_flow_integrity [Professional Edition only] [V2.08.00 or later]	This option generates code for the detection of illegal indirect function calls.
List Options	-listfile	A source list file is output.
	-nolistfile	A source list file is not output.
	-show	Specifies the contents of the source list file.

Classification	Option	Description
Optimize Options (1/2)	-optimize	Selects the optimization level.
	-goptimize	Outputs additional information for inter-module optimization.
	-speed	Optimization is with emphasis on execution performance.
	-size	Optimization is with emphasis on code size.
	-loop	Specifies a maximum number for loop-expansion.
	-inline	Inline expansion is processed automatically.
	-noinline	Inline expansion is not processed automatically.
	-file_inline	Specifies a file for inter-file inline expansion.
	-case	Selects the method of expansion for **switch** statements.
	-volatile	External variables are handled as if they are all **volatile** qualified.
	-novolatile	External variables are handled as if none of them have been declared **volatile**.
	-type_size_access_to_volatile [V3.04.00 or later]	A variable with volatile specified is accessed in the size of the variable type.
	-const_copy	Enables constant propagation of **const** qualified external variables.
	-noconst_copy	Disables constant propagation of **const** qualified external variables.
	-const_div	Divisions and remainders of integer constants are converted into instruction sequences.
	-noconst_div	Divisions and remainders of integer constants are not converted into instruction sequences.
	-library	Selects the method for the execution of library functions.
	-scope	Selects division of the ranges for optimization into multiple sections before compilation.
	-noscope	Selects non-division of the ranges for optimization into multiple sections before compilation.
	-schedule	Pipeline processing is considered in scheduling instructions.
	-noschedule	Scheduling is not applied to instruction execution.
	-map	All access to external variables is optimized.
	-smap	Access to external variables is optimized as defined in the file to be compiled.
	-nomap	Access to external variables is not optimized.
	-approxdiv	Division of floating-point constants is converted into multiplication.
	-enable_register	Variables with the **register** storage class specification are given preference for allocation to registers.

Classification	Option	Description
Optimize Options (2/2)	-simple_float_conv	Part of the type conversion processing between the floating-point type and the integer type is omitted.
	-fpu	Single-precision floating-point processing instructions are used.
	-nofpu	Single-precision floating-point processing instructions are not used.
	-dpfpu [V3.01.00 or later]	Double-precision floating-point processing instructions are used.
	-nodpfpu [V3.01.00 or later]	Double-precision floating-point processing instructions are not used.
	-tfu [V3.01.00 or later]	Selects how the trigonometric function unit is to be used.
	-tfu_version [V3.05.00 or later]	Selects the version of the trigonometric function unit.
	-nosave_tfu [V3.05.00 or later]	A code for saving and restoring the output of the trigonometric function unit (v2) is not generated for interrupt functions.
	-alias	Optimization is performed in consideration of the types of data indicated by pointers.
	-float_order	The orders of operations in floating-point expressions are modified for optimization.
	-branch_chaining [V3.03.00 or later]	The branch instruction size is reduced for optimization.
	-nobranch_chaining [V3.03.00 or later]	The branch instruction size is not reduced for optimization.
	-ip_optimize	Selects global optimization.
	-merge_files	The results of compiling multiple source files are output to a single object file.
	-whole_program	Makes the compiler perform optimization on the assumption that all source files have been input.

Classification	Option	Description
Microcontroller Options	-isa	Selects the instruction-set architecture.
	-cpu	Selects the microcontroller type.
	-endian	Selects the endian type.
	-round	Selects the rounding method for floating-point constant operations.
	-denormalize	Selects the operation when denormalized numbers are used to describe floating-point constants.
	-dbl_size	Selects the precision of the **double** and **long double** types.
	-int_to_short	Replaces the **int** type with the **short** type and the **unsigned int** type with the **unsigned short** type.
	-signed_char	Variables of the **char** type are handled as **signed char**.
	-unsigned_char	Variables of the **char** type are handled as **unsigned char**.
	-signed_bitfield	The sign bits of bit-fields are taken as **signed**.
	-unsigned_bitfield	The sign bits of bit-fields are taken as **unsigned**.
	-auto_enum	Selects whether or not the sizes for enumerated types are automatically selected.
	-bit_order	Selects the order of bit-field members.
	-pack	Specifies one as the boundary alignment value for structure members and class members.
	-unpack	Aligns structure members and class members to the alignment boundaries for the given data types.
	-exception	Enables the exception handling function.
	-noexception	Disables the exception handling function.
	-rtti	Selects enabling or disabling of C++ runtime type information (**dynamic_cast** or **typeid**).
	-fint_register	Selects a general register for exclusive use with the fast interrupt function.
Microcontroller Options	-branch	Selects the maximum size or no maximum size for branches.
	-base	Specifies the base registers for ROM and RAM.
	-patch	Selects avoidance or non-avoidance of a problem specific to the CPU type.
	-pic	Enables the PIC function.
	-pid	Enables the PID function.
	-nouse_pid_register	The PID register is not used in code generation.
	-save_acc	The contents of ACC are saved and restored in interrupt functions.

Classification	Option	Description
Assemble and Linkage Options	-asmcmd	Specifies a subcommand file for **asrx** options.
	-lnkcmd	Specifies a subcommand file for **rlink** options.
	-asmopt	Specifies **asrx** options.
	-lnkopt	Specifies **rlink** options.
Other Options	-logo	Selects the output of copyright information.
	-nologo	Selects the non-output of copyright information.
	-euc	The character codes of input programs are interpreted as EUC codes.
	-sjis	The character codes of input programs are interpreted as SJIS codes.
	-latin1	The character codes of input programs are interpreted as ISO-Latin1 codes.
	-utf8	The character codes of input programs are interpreted as UTF-8 codes.
	-big5	The character codes of input programs are interpreted as BIG5 codes.
	-gb2312	The character codes of input programs are interpreted as GB2312 codes.
	-outcode	Selects the character coding for an output assembly-language file.
	-subcommand	Specifies a file for including command options.

Source Options

< Compile Options / Source Options >

The following source options are available.

- -lang
- -include
- -preinclude
- -define
- -undefine
- -message
- -nomessage
- -change_message
- -no_warning [V2.08.00 or later]
- -file_inline_path
- -comment
- -truncated_address_initializer [V3.01.00 or later]
- -check
- -misra2004 [Professional Edition only]
- -misra2012 [Professional Edition only] [V2.04.00 or later]
- -ignore_files_misra [Professional Edition only]
- -check_language_extension [Professional Edition only]
- -misra_intermodule [Professional Edition only] [V3.01.00 or later]

-lang

< Compile Options / Source Options >

[Format]

```
-lang= { c | cpp | ecpp | c99 }
```

- [Default]
  If this option is not specified, the compiler will compile the program file as a C++ source file when the extension is **cpp**, **cc**, or **cp**, and as a C (C89) source file for any other extensions. However, if the extension is **src** or **s**, the program file is handled as an assembly-language file regardless of whether this option is specified.

[Description]

- This option specifies the language of the source file.
- When the **lang=c** option is specified, the compiler will compile the program file as a C (C89) source file.
- When the **lang=cpp** option is specified, the compiler will compile the program file as a C++ source file.
- When the **lang=ecpp** option is specified, the compiler will compile the program file as an Embedded C++ source file.
- When the **lang=c99** option is specified, the compiler will compile the program file as a C (C99) source file.

[Remarks]

- The Embedded C++ language specification does not support a **catch**, **const_cast**, **dynamic_cast**, **explicit**, **mutable**, **namespace**, **reinterpret_cast**, **static_cast**, **template**, **throw**, **try**, **typeid**, **typename**, **using**, multiple inheritance, or virtual base class. If one of these classes is written in the source file, the compiler will display an error message.
- Always specify the **lang=ecpp** option when using an EC++ library.
- In batch compilation (when multiple C/C++ language source files are input to the compiler at the same time), the individual C or C++ language source files must be in the same language. Thus, separate the C and C++ language source files in accord with the languages to be specified and then perform batch compilation by specifying this option for the group in each of the languages.

-include

< Compile Options / Source Options >

[Format]

```
-include=<path name>[,...]
```

[Description]

- This option specifies the name of the path to the folder that stores the include file.
- Multiple path names can be specified by separating them with a comma (,).
- Searching for files with names enclosed in "<" and ">" proceeds in order of the folders specified by the **include** option, the folders specified by environment variable **INC_RX**.
- Searching for files with names enclosed in double quotation marks ("") proceeds in order of the storage folder of the file for which the #include statement is made, the folders specified by the **include** option, the folders specified by environment variable **INC_RX**.
- If two or more folders are specified in the **include** option, searching proceeds in order of specifications of the pathnames for the folders on the command line (from left to right).

[Remarks]

- If this option is specified for more than one time, all specified path names are valid.

-preinclude

< Compile Options / Source Options >

[Format]

```
-preinclude=<file name>[,...]
```

[Description]

- This option includes the specified file contents at the head of the compiling unit.
- If the file name is specified by its relative path, the folder is searched in the following order:
    - [V3.01.00 or earlier]
        - Folder with the compilation unit
        - Folder specified by the include option
        - Folder specified by environment variable **INC_RX**
    - [V3.02.00 or later]
        - Folder that started the compiler
- Multiple file names can be specified by separating them with a comma (,).

[Remarks]

- If this option is specified for more than one time, all specified files will be included.

-define

< Compile Options / Source Options >

[Format]

```
-define=<sub>[,...]
        <sub>: <macro name> [= <string>]
```

[Description]

- This option provides the same function as **#define** specified in the source file.
- <string> can be defined as a macro name by specifying <macro name>=<string>.
- When only <macro name> is specified as a suboption, the macro name is assumed to be defined. (An empty value is defined rather than 1.)
- Names or integer constants can be written in <string>.

[Remarks]

- If the macro name specified by this option has already been defined in the source file by **#define**, **#define** takes priority.
- If this option is specified for more than one time, all specified macro names are valid.

-undefine

< Compile Options / Source Options >

[Format]

```
-undefine=<sub>[,...]
          <sub>: <macro name>
```

[Description]

- This option invalidates the predefined macro of <macro name>.
- Multiple macro names can be specified by separating them with a comma (,).

[Remarks]

- Refer to the Macro Names section, in the COMPILER LANGUAGE SPECIFICATIONS chapter, for specifiable predefined macros of the compiler.
- If this option is specified for more than one time, all specified macro names will be undefined.

-message

< Compile Options / Source Options >

[Format]

```
-message
```

- [Default]
  No information-level messages will be output.

[Description]

- This option outputs the information-level messages.

[Remarks]

- Message output from the assembler or optimizing linkage editor cannot be controlled by this option. Message output from the optimizing linkage editor can be controlled by using the **lnkcmd** option to specify the **message** or **nomessage** option of the optimizing linkage editor.

-nomessage

< Compile Options / Source Options >

[Format]

```
-nomessage [= <error number> [- <error number>][,...]
```

[Description]

- With no sub-options, this disables the output of information-level messages.
- When error numbers are specified as sub-options, only the output of the information-level messages with the specified numbers will be disabled. Other information-level messages will be output.
- A range of error numbers to be disabled can be specified by using a hyphen (-), that is, in the form of <error number>-<error number>.
- Error numbers are specified by the five lower-order digits (i.e. five digits from the right) of message numbers without the prefix "M" (information).
  Example: To change the level of information message M0523009
  -nomessage=23009

[Remarks]

- Message output from the assembler or optimizing linkage editor cannot be controlled by this option. Message output from the optimizing linkage editor can be controlled by using the **lnkcmd** option to specify the **message** or **nomessage** option of the optimizing linkage editor.
- If the **nomessage** option is specified for more than one time, output for all specified error numbers will be disabled.
- This option is only specifiable for messages with number 0510000 to 0549999 (including the component number).
- This option can also be used to suppress the output of warnings in the range from 0520000 to 0529999. Specify the numbers of the warnings that you wish to suppress. To suppress the output of other warnings, use **-change_message** to change them to information-level messages.

-change_message

< Compile Options / Source Options >

[Format]

```
-change_message = <sub>[,...]
                  <sub>: <error level>[=<error number>[- <error number>][,...]]
                         <error level>: { information | warning | error }
```

[Description]

- This option changes the message level of information-level and warning-level messages.
- Multiple error numbers can be specified by separating them with a comma (,).
- Error numbers are specified by the five lower-order digits (i.e. five digits from the right) of the message numbers without the prefix "M" (information) or "W" (warning).
  Example: To change the level of information message M0523009
  -change_message=error=23009
- Although this option may change the types of some messages (e.g. error (E) or warning (W)), the meaning of the message indicated by the component or message number remains the same.

[Example]

```
change_message=information=error number
```

- Warning-level messages with the specified error numbers are changed to information-level messages.

```
change_message=warning=error number
```

- Information-level messages with the specified error numbers are changed to warning-level messages.

```
change_message=error=error number
```

- Information-level and warning-level messages with the specified error numbers are changed to error-level messages.

```
change_message=information
```

- All warning-level messages are changed to information-level messages.

```
change_message=warning
```

- All information-level messages are changed to warning-level messages.

```
change_message=error
```

- All information-level and warning-level messages are changed to error-level messages.

[Remarks]

- The output of messages which have been changed to information-level messages can be disabled by the **nomessage** option.
- Message output from the assembler or optimizing linkage editor cannot be controlled by this option. Message output from the optimizing linkage editor can be controlled by using the **lnkcmd** option to specify the **message** or **nomessage** option of the optimizing linkage editor.
- If this option is specified for more than one time, all specified error numbers are valid.

- Only the levels of warning and information messages can be controlled by this option. Specification of the option for a message not at these levels is ignored.
- This option is not usable to control the level of MISRA2004 detection messages (labeled M) that appear when the **misra2004** option has been specified.

-no_warning [V2.08.00 or later]

< Compile Options / Source Options >

[Format]

```
-no_warning={ <number> | <number>-<number> }[,{ <number> | <number>-<number> }]...
```

[Description]

- This option is used to suppress the output of warnings or information-level messages with the numbers specified by <**number**>. You cannot use this option to suppress the output of any messages by the assembler or optimizing linkage editor.
- You can specify this option more than once. Each specification will be effective.
- Separate all parameters with commas (,). Do not enter any white space before or after a comma.
- Entering white space before or after a comma, omitting parameters, or entering characters that are not numbers will lead to a compilation error.
- The values specifiable as parameters are the last five digits of message numbers of warnings and information-level messages (51000 to 54999). Any other specified numbers, including those that are not actually message numbers, will be ignored.
    - Specification of the numbers of error messages will be ignored because the output of error messages cannot be suppressed. This applies to any messages at the error level, including those having levels changed by the **-change_message=error** option.
- This option is also usable in combination with the **-nomessage** option. In such cases, both options are effective, even if they include specifications of the same message.

-file_inline_path

< Compile Options / Source Options >

[Format]

```
-file_inline_path=<path name>[,...]
```

[Description]

- This option is not available in V.2.00. Any specification of this option will simply be ignored and will not lead to an error due to compatibility with former versions.

-comment

< Compile Options / Source Options >

[Format]

```
-comment = { nest | nonest }
```

[Description]

- When **comment=nest** is specified, nested comments are allowed to be written in the source file.
- The default for this option is **comment=nonest**.
- When **comment=nonest** is specified, comment nesting is not recognized.

[Example]

- When **comment=nest** is specified, the compiler handles the above line as a nested comment; however, when **comment=nonest** is specified, the compiler assumes (1) as the end of the comment.

```
/* This is an example of /* nested */ comment */
                                  (1)
```

-truncated_address_initializer [V3.01.00 or later]

< Compile Options / Source Options >

[Format]

```
-truncated_address_initializer
```

[Description]

- The warning message W0520069 is output instead of error E0520069 in response to detection of a code that initializes a 1-byte type or 2-byte type external variable or static variable by an address value. This is only applicable to C-language source files.

[Remarks]

- The upper bytes of an address are truncated. Therefore, the original address value will not be retained when the upper bytes are not 0.
- If casting into 1-byte or 2-byte type is followed by casting into a larger type, an error will occur.
- If the variable to be initialized is a bit field member, an error will occur.
- This option is invalid when code is in the C++ language or Embedded C++ language.

-check

< Compile Options / Source Options >

[Format]

```
-check = { nc | ch38 | shc }
```

[Description]

- This option checks the specified options and source file parts which will affect the compatibility when this compiler uses a C/C++ source file that has been coded for the R8C and M16C family C compilers, H8, H8S, and H8SX family C/C++ compilers, and SuperH family C/C++ compilers.
- For **check=nc**, the compatibility with the R8C and M16C family C compilers is checked. Checking will be for the following options and types:
- Options: signed_char, signed_bitfield, bit_order=left, endian=big, and dbl_size=4
- inline, enum type, #pragma BITADDRESS, #pragma ROM, #pragma PARAMETER, and asm()
- Assignment of a constant outside the **signed short** range to the **int** or **signed int** type or assignment of a constant outside the **unsigned short** range to the **int** or **unsigned int** type while **-int_to_short** is not specified
- Assignment of a constant outside both of the **signed short** and **unsigned short** ranges to the **long** or **long long** type
- Comparison expression between a constant outside the **signed short** range and the **int**, **short**, or **char** type (except the **signed char** type)
- For **check=ch38**, the compatibility with the H8, H8S, and H8SX family C/C++ compilers is checked. Checking will be for the following options and types:
- Options: unsigned_char, unsigned_bitfield, bit_order=right, endian=little, and dbl_size=4
- __asm and #pragma unpack
- Comparison expression with a constant greater than the maximum value of **signed long**
- Assignment of a constant outside the **signed short** range to the **int** or **signed int** type or assignment of a constant outside the **unsigned short** range to the **int** or **unsigned int** type while **-int_to_short** is not specified
- Assignment of a constant outside both of the **signed short** and **unsigned short** ranges to the **long** or **long long** type
- Comparison expression between a constant outside the signed short range and the **int**, **short**, or **char** type (except the **signed char** type)
- For **check=shc**, the compatibility with the SuperH family C/C++ compilers is checked. Checking will be for the following options and types:
- Options: unsigned_char, unsigned_bitfield, bit_order=right, endian=little, dbl_size=4, and round=nearest
- #pragma unpack
- **volatile** qualified variables
- Confirm the following notes for the displayed items.
- Options: The settings which are not defined in the language specification and depend on implementation differ in each compiler. Confirm the settings of the options that were output in a message.
- Extended specifications: There is a possibility that extended specifications will affect program operation. Confirm the descriptions on the extended specifications that were output in a message.

[Remarks]

- When **dbl_size=4** is enabled, the results of type conversion related to floating-point numbers and the results of library calculation may differ from those in the R8C and M16C family C compilers, H8, H8S, and H8SX family C/C++ compilers, and SuperH family C/C++ compilers. When **dbl_size=4** is specified, this compiler handles **double** type and **long double** type as 32 bits, but the R8C and M16C family C compilers (**fdouble_32**), H8, H8S, and H8SX family C/C++ compilers (**double=float**), and SuperH family C/C++ compilers (**double=float**) handle only **double** type as 32 bits.
- The result of a binary operation (addition, subtraction, multiplication, division, comparison, etc.) with **unsigned int** type and **long** type operands may differ from that in the SuperH family C/C++ compilers. In this compiler, the types of the operands are converted to the **unsigned long** type before operation. However, in the SuperH family C/C++ compilers (only when **strict_ansi** is not specified), the types of the operands are converted to the **signed long long** type before operation.
- The data size of reading from and writing to a **volatile** qualified variable may differ from that in the SuperH family C/C++ compilers. This is because a **volatile** qualified bit field may be accessed in a size smaller than that of the declaration type in this compiler. However, in the SuperH family C/C++ compilers, a **volatile** qualified bit field is accessed in the same size as that of the declaration type.
- This option does not output a message regarding allocation of structure members and bit field members. When an allocation-conscious declaration is made, refer to the Internal Data Representation and Areas section of the COMPILER LANGUAGE SPECIFICATIONS chapter.
- In the R8C and M16C family C compilers (**fextend_to_int** is not specified), the generated code has been evaluated without performing generalized integer promotion by a conditional expression. Accordingly, operation of such a code may differ from a code generated by this compiler.

-misra2004 [Professional Edition only]

< Compile Options / Source Options >

[Format]

```
-misra2004 = {
   all
   | apply=<rule number>[,...]
   | ignore=<rule number>[,...]
   | required
   | required_add=<rule number>[,...]
   | required_remove=<rule number>[,...]
   | <filename> }
```

[Description]

- This option enables checking against the MISRA-C:2004 rules and to select specific rules to be used.
- When **misra2004=all**, the compiler checks the source code against all of the rules that are supported.
- When **misra2004=apply=<rule number>[,<rule number>,...]**, the compiler checks the source code against the rules with the selected numbers.
- When **misra2004=ignore=<rule number>[,<rule number>,...]**, the compiler checks the source code against the rules other than those with the selected numbers.
- When **misra2004=required**, the compiler checks the source code against the rules of the "required" type.
- When **misra2004=required_add=<rule number>[,<rule number>,...]**, the compiler checks the source code against the rules of the "required" type and the rules with the selected numbers.
- When **misra2004=required_remove=<rule number>[,<rule number>,...]**, the compiler checks the source code against the rules other than those with the selected numbers among the rules of the "required" type.
- When **misra2004=<filename>**, the compiler checks the source code against the rules with the numbers written in the specified file. One rule number is written per line in the file. Each rule number must be specified by using decimal values and a period (".").
- When checking of a line of code against the MISRA-C:2004 rules leads to detection of a violation, a message in the following format will appear.
  <Filename> (<line number>): M0523028 <Rule number>: <Message>
- When **-misra2004=<filename>** is used more than once, only the last specification is valid.

[Remarks]

- The -misra2004 option can be specified more than once. However, if multiple types exist, only the type written last and consecutive specifications of the same type are valid.
  ... -misra2004=ignore=2.2 -misra2004=apply=2.3
  -misra2004=required_add=4.1 -misra2004=apply=4.2
  -misra2004=apply=5.2 ...
  In this example, **ignore**, **apply**, and **required_add** are specified, but only **apply** (used in the last two cases) is valid. The compiler will check the source code against rules 4.2 and 5.2.
- When the number of an unsupported rule is specified for **<rule number>**, the compiler detects error F0523031 and stops the processing.
- When the file specified in **misra2004=<filename>** cannot be opened, the compiler detects error F0523029. When rule numbers are not extractable from the specified file, the compiler detects error F0523030. Processing by the compiler stops in both cases.
- This option is ignored when cpp, c99, or ecpp is selected for the **lang** option or when **output=prep** is specified at the same time.
- This option supports the MISRA-C: 2004 rules listed below.

[Required]
2.2 2.3
4.1 4.2
5.2 5.3 5.4
6.1 6.2 6.4 6.5
7.1
8.1 8.2 8.3 8.5 8.6 8.7 8.11 8.12
9.1 9.2 9.3
10.1 10.2 10.3 10.4 10.5 10.6
11.1 11.2 11.5
12.3 12.4 12.5 12.7 12.8 12.9 12.10 12.12
13.1 13.3 13.4
14.2 14.3 14.4 14.5 14.6 14.7 14.8 14.9 14.10
15.1 15.2 15.3 15.4 15.5
16.1 16.3 16.5 16.6 16.9
18.1 18.4
19.3 19.6 19.8 19.11 19.14 19.15
20.4 20.5 20.6 20.7 20.8 20.9 20.10 20.11 20.12
[Not required]
5.5 5.6
6.3
11.3 11.4
12.1 12.6 12.11 12.13
13.2
17.5
19.7 19.13

- For source programs that use extended functions such as **#pragma**, checking against these rules will be suppressed under some conditions. For details, refer to the section on the **check_language_extension** option.
- MISRA diagnostic messages displayed by the **misra2004** option cannot be controlled by the **change_message** option.
- The source code cannot be simultaneously checked against the MISRA-C: 2012 rules.

-misra2012 [Professional Edition only] [V2.04.00 or later]

< Compile Options / Source Options >

[Format]

```
-misra2012 = {
   all
   | apply=<rule number>[,...]
   | ignore=<rule number>[,...]
   | required
   | required_add=<rule number>[,...]
   | required_remove=<rule number>[,...]
   | <filename> }
```

[Description]

- This option enables checking against the MISRA-C:2012 rules and to select specific rules to be used.
- When **misra2012=all**, the compiler checks the source code against all of the rules that are supported.
- When **misra2012=apply=<rule number>[,<rule number>,...]**, the compiler checks the source code against the rules with the selected numbers.
- When **misra2012=ignore=<rule number>[,<rule number>,...]**, the compiler checks the source code against the rules other than those with the selected numbers.
- When **misra2012=required**, the compiler checks the source code against the rules of the "mandatory" and "required" types.
- When **misra2012=required_add=<rule number>[,<rule number>,...]**, the compiler checks the source code against the rules of the "mandatory" and "required" types and the rules with the selected numbers.
- When **misra2012=required_remove=<rule number>[,<rule number>,...]**, the compiler checks the source code against the rules other than those with the selected numbers among the rules of the "required" type.
- When **misra2012=<filename>**, the compiler checks the source code against the rules with the numbers written in the specified file. One rule number is written per line in the file. Each rule number must be specified by using decimal values and a period (".").
- When checking of a line of code against the MISRA-C:2012 rules leads to detection of a violation, a message in the following format will appear.
  <Filename> (<line number>): M0523086 <Rule number>: <Message>
- When **-misra2012=<filename>** is used more than once, only the last specification is valid.

[Remarks]

- The -misra2012 option can be specified more than once. However, if multiple types exist, only the type written last and consecutive specifications of the same type are valid.
  ... -misra2012=ignore=3.1
  -misra2012=required_add=4.1 -misra2012=apply=4.2
  -misra2012=apply=5.2 ...
  In this example, **ignore**, **apply**, and **required_add** are specified, but only **apply** (used in the last two cases) is valid. The compiler will check the source code against rules 4.2 and 5.2.
- When the number of an unsupported rule is specified for **<rule number>**, the compiler detects error F0523031 and stops the processing.
- When the file specified in **misra2012=<filename>** cannot be opened, the compiler detects error F0523029. When rule numbers are not extractable from the specified file, the compiler detects error F0523030. Processing by the compiler stops in both cases.
- This option is ignored when cpp or ecpp is selected for the **lang** option or when **output=prep** is specified at the same time.

- If this option is specified together with **lang=c99**, checking against C90/C99 common rules is performed in the scope of C99.
- For source programs that use extended functions such as **#pragma**, checking against these rules are suppressed under some conditions. For details, refer to the section on the **check_language_extension** option.
- MISRA diagnostic messages displayed by the **misra2012** option cannot be controlled by the **change_message** option.
- Checking the source code cannot be simultaneously checked against the MISRA-C:2004 rules.
- In V3.02.00 and later versions, this option supports the MISRA-C: 2012 rules listed below.
  2.2 2.6 2.7
  3.1 3.2
  4.1 4.2
  5.1(*2) 5.2 5.3 5.4 5.5 5.6(*2) 5.7(*2) 5.8(*2) 5.9(*2)
  6.1 6.2
  7.1 7.2 7.3 7.4
  8.1 8.2 8.3(*2) 8.4 8.5(*3) 8.6(*3) 8.8 8.9 8.11 8.12 8.13 8.14
  9.1 9.2 9.3 9.4 9.5
  10.1 10.2 10.3 10.4 10.5 10.6 10.7 10.8
  11.1 11.2 11.3 11.4 11.5 11.6 11.7 11.8 11.9
  12.1 12.2 12.3 12.4 12.5
  13.1 13.2 13.3 13.4 13.5 13.6
  14.2 14.3 14.4
  15.1 15.2 15.3 15.4 15.5 15.6 15.7
  16.1 16.2 16.3 16.4 16.5 16.6 16.7
  17.1 17.3 17.4 17.5 17.6 17.7 17.8
  18.4 18.5 18.7
  19.2
  20.1 20.2 20.3 20.4 20.5 20.6 20.7 20.8 20.9 20.10 20.11 20.12 20.13 20.14
  21.1 21.2 21.3 21.4 21.5 21.6 21.7 21.8(*1) 21.9 21.10 21.11 21.12

  *1) Checking is in accord with to MISRA C:2012, Amendment 1 (getenv is not checked).
  *2) If the **-misra_intermodule** option is specified, analysis can be performed over multiple files.
  *3) This rule is valid only when the **-misra_intermodule** option is specified.

[Example]

- In this example, tree types of **-misra2012** option (**ignore**, **apply,** and **required_add**) are specified, but only **apply** in the last two consecutive specifications is valid. As a result, the compiler checks the source code against rules 4.2 and 5.2.

```
-misra2012=ignore=3.1
-misra2012=required_add=4.1 -misra2012=apply=4.2
-misra2012=apply=5.2
```

-ignore_files_misra [Professional Edition only]

< Compile Options / Source Options >

[Format]

```
-ignore_files_misra=<filename>[,<filename>,...]
```

[Description]

- This option selects source files that will not be checked against the MISRA-C:2004 rules or MISRA-C:2012 rules.

[Remarks]

- If the option is specified more than once in the command line, all specifications are valid.
- This option is ignored when the **misra2004** or **misra2012** option has not been specified.
- <filename> is ignored when the specified file is not to be compiled.

-check_language_extension [Professional Edition only]

< Compile Options / Source Options >

[Format]

```
-check_language_extension
```

[Description]

- This option enables complete checking against the MISRA-C: 2004 rules or MISRA-C: 2012 rules for parts of the code where they would otherwise be suppressed due to proprietary extensions from the C language specification.
- With the default **misra2004** option and **misra2012** option, the compiler does not proceed with checking against the MISRA-C: 2004 rules and MISRA-C: 2012 rules under the condition given below. To enable complete checking, specify the check_language_extension option when specifying the **misra2004** option or **misra2012** option.
    - A function has no prototype declaration (MISRA-C:2004 rule 8.1, MISRA-C:2012 rule 8.4) and **#pragma entry** or **#pragma interrupt** is specified for it.

[Example]

```
#pragma interrupt vfunc
extern void service(void);
void vfunc(void)
{
    service();
}
```

- A function vfunc, for which **#pragma interrupt** is specified, has no prototype declaration. The message on rule 8.1 is not displayed unless the **check_language_extension** option is specified.

[Remarks]

This option is ignored when the **misra2004** or **misra2012** option has not been specified.

-misra_intermodule [Professional Edition only] [V3.01.00 or later]

< Compile Options / Source Options >

[Format]

```
-misra_intermodule=<file name>
```

- Interpretation when omitted
  None (checking of source code in multiple files against the MISRA-C:2012 rules is disabled)

[Description]

- This option saves symbol information of multiple files in <file name> and checks source code in these files against the MISRA-C:2012 rules. If <file name> does not exist, a new file will be created. If <file name> exists, symbol information will be added to the file.
- This option is only valid when the **-misra2012** option is specified. A warning is output and this option will be ignored if the **-misra2012** option is not specified.
- An error will occur if <file name> is omitted.
- This option is applied to rules whose valid range of analysis is the system. Source code will be checked against the following MISRA-C:2012 rules.
  5.1 5.6 5.7 5.8 5.9
  8.3 8.5 8.6

[Example]

- To check source code in multiple files a.c, b.c, and c.c against the MISRA-C:2012 rules, describe as:

```
> ccrx -isa=rxv3 -misra2012=all -misra_intermodule=test.mi a.c b.c c.c
```

[Remarks]

- .{c|a|f} cannot be specified as the extension of <file name>. If specified, an error will occur. Correct operation is not guaranteed if <file name> is same as the name of another input or output file.
- If there are many files to be checked and the symbol information to be stored in <file name> is huge, the compilation speed gets slower.
- If any of the source files is modified after <file name> was created, recompilation will update the information of <file name>. If any of the source files is deleted or its file name is changed, delete <file name> and recheck source code against the MISRA-C:2012 rules because the information of <file name> cannot be updated.
- An error will occur if this option is specified in the Standard edition of the compiler.
- This option cannot correctly check the source code when files are compiled in parallel by using, for example, parallel builds. Specify this option without performing parallel compilation.

Object Options

< Compile Options / Object Options >

The following object options are available.

- -output
- -noline
- -debug
- -nodebug
- -g_line [V3.02.00 or later]
- -section
- -stuff
- -nostuff
- -instalign4
- -instalign8
- -noinstalign
- -nouse_div_inst
- -create_unfilled_area [V2.03.00 or later]
- -stack_protector/-stack_protector_all [Professional Edition only] [V2.04.00 or later]
- -avoid_cross_boundary_prefetch [V2.07.00 or later]
- -insert_nop_with_label [V2.08.00 or later]
- -control_flow_integrity [Professional Edition only] [V2.08.00 or later]

-output

< Compile Options / Object Options >

[Format]

```
-output = <sub> [=<file name>]
          <sub>: { prep | src | obj | abs | hex | sty }
```

- [Default]
  The default for this option is **output=obj**.

[Description]

- This option specifies the output file type.
- The suboptions and output files are shown in the following table.
- If no **<file name>** is specified, a file will be generated with an extension, that is shown in the following table, appended to the source file name input at the beginning.

Table 2.3 Suboption Output Format

Suboption	Output File Type	Extension When File Name is Not Specified
prep	Source file after preprocessed	C (C89, C99) source file: p C++ source file: pp
src	Assembly-language file	src
obj	Relocatable file	obj
abs	Absolute file	abs
hex	Intel hex type file	hex
sty	Motorola S type file	mot

Note Relocatable files are files output from the assembler.
Absolute files, Intel hex type files, and Motorola S type files are files output from the optimizing linkage editor.

[Remarks]

- An intermediate file used to generate a file of the specified type is not generated.

-noline

< Compile Options / Object Options >

[Format]

```
-noline
```

[Description]

- This option disables **#line** output during preprocessor expansion.

[Remarks]

- This option is validated when the **output=prep** option has not been specified.

-debug

< Compile Options / Object Options >

[Format]

```
-debug
```

- [Default]
  The default for this option is **-nodebug**.

[Description]

- When the **debug** option is specified, debugging information necessary for C-source debugging is output. The **debug** option is valid even when an optimize option is specified.

-nodebug

< Compile Options / Object Options >

[Format]

```
-nodebug
```

- [Default]
  The default for this option is **-nodebug**.

[Description]

- When the **nodebug** option is specified, no debugging information is output.

-g_line [V3.02.00 or later]

< Compile Options / Object Options >

[Format]

```
-g_line
```

[Description]

- This option is valid only when the **-debug** option is specified at the same time.
- This option enhances debugging information to enable more accurate single-step execution at the source level during debugging when optimization is performed.
- An increase in debugging information might cause a delay in single-step execution.

[Example]

```
ccrx a.c -isa=rxv3 -debug -g_line
```

-section

< Compile Options / Object Options >

[Format]

```
-section = <sub>[,...]
           <sub>: { P = <section name> |
                    C = <section name> |
                    D = <section name> |
                    B = <section name> |
                    L = <section name> |
                    W = <section name> }
```

[Description]

- This option specifies the section name.
- **section=P=<section name>** specifies the section name of a program area.
- **section=C=<section name>** specifies the section name of a constant area.
- **section=D=<section name>** specifies the section name of an initialized data area.
- **section=B=<section name>** specifies the section name of an uninitialized data area.
- **section=L=<section name>** specifies the section name of a literal area.
- **section=W=<section name>** specifies the section name of a **switch** statement branch table area.
- <section name> must be alphabetic, numeric, underscore (_), or $. The first character must not be numeric.

[Remarks]

- The default for this option is **section=P=P,C=C,D=D,B=B,L=L,W=W**.
- In the same way as in V. 1.00, if you want to output the literal area in the **C** section rather than output a separate **L** section, select **section=L=C**.
- Except for changing the **L** section to the same section name as that of the **C** section, the same section name cannot be specified for the sections for different areas.
- For the translation limit of the section name length, refer to Translation Limits.

-stuff

< Compile Options / Object Options >

[Format]

```
-stuff
```

[Description]

- When the **stuff** option is specified, all variables are allocated to 4-byte, 2-byte, or 1-byte boundary alignment sections depending on the alignment value (see Table 2.4).
- [V3.01.00 or later] When the **dpfpu** option is specified, double type and long double type variables are allocated to 8-byte boundary alignment sections.

Table 2.4 Correspondences between Variables and Their Output Sections When **stuff** Option is Specified

Variable Type	Alignment Value for Variable	Section to Which Variable Belongs
const qualified variables	4	C_8*1 C*2
	2	C_2
	1	C_1
Initialized variables	4	D_8*1 D*2
	2	D_2
	1	D_1
Uninitialized variables	4	B_8*1 B*2
	2	B_2
	1	B_1
switch statement branch table	4	W
	2	W_2
	1	W_1

Notes 1. When the variable type is double or long double with the **dpfpu** option specified

Notes 2. Cases other than note 1

- **C**, **D**, and **B** are the section names specified by the **section** option or **#pragma section**. **W** is the section name specified by the **section** option. The data contents allocated to each section are output in the order they were defined.

[Example]

```
int a;
char b=0;
const short c=0;
struct {
        char x;
        char y;
} ST;
```

```
        .SECTION                C_2,ROMDATA,ALIGN=2
        .glb        _c
_c:
        .word                   0000H
        .SECTION                D_1,ROMDATA
        .glb        _b
_b:
        .byte                   00H
        .SECTION                B,DATA,ALIGN=4
        .glb        _a
_a:
        .blkl        1
        .SECTION                B_1,DATA
        .glb        _ST
_ST
        .blkb        2
```

[Remarks]

- The **-stuff** option has no effect for sections other than **B**, **D**, **C**, and **W**.

-nostuff

< Compile Options / Object Options >

[Format]

```
-nostuff [= <section type>[,...]]
            <section type>: { B | D | C | W }
```

[Description]

- When the **nostuff** option is specified, the compiler allocates the variables belonging to the specified <section type> to 4-byte boundary alignment sections. When **<section type>** is omitted, variables of all section types are applicable.
- **C**, **D**, and **B** are the section names specified by the **section** option or **#pragma** section. **W** is the section name specified by the section option. The data contents allocated to each section are output in the order they were defined.

[Example]

```
int a;
char b=0;
const short c=0;
struct {
        char x;
        char y;
} ST;
```

```
        .SECTION                C,ROMDATA,ALIGN=4
        .glb        _c
_c:
        .word                   0000H
        .SECTION                D,ROMDATA,ALIGN=4
        .glb        _b
_b:
        .byte                   00H
        .SECTION                B,DATA,ALIGN=4
        .glb        _a
_a:
        .blkl        1

        .glb        _ST
_ST
        .blkb        2
```

[Remarks]

- The **nostuff** option cannot be specified for sections other than **B**, **D**, **C**, and **W**.

-instalign4

< Compile Options / Object Options >

[Format]

```
-instalign4[={loop|inmostloop}]
```

[Description]

- This option aligns instructions at branch destinations.
- When the **instalign4** option is specified, the instruction at the location address is aligned to the 4-byte boundary.
- Instruction alignment is performed only when the instruction at the specified location exceeds the address which is a multiple of the alignment value (4)*1.
- The following three types of branch destination can be selected by specifying the suboptions of **-instalign4***2.
- No specification:Head of function and **case** and **default** labels of **switch** statement
  inmostloop: Head of each inmost loop, head of function, and **case** and **default** labels of **switch** statement
  loop: Head of each loop, head of function, and **case** and **default** labels of **switch** statement
- When this option is selected, the alignment value of the program section is changed from 1 to 4 (for **instalign4**) or 8 (for **instalign8**).
- This option aims to efficiently operate the instruction queues of the RX CPU and improve the speed of program execution by aligning the addresses of branch destination instructions.
- This option is designed for improving the performance for the devices of the following groups:
  RX110, RX111, RX113, RX130, RX13T, RX210, RX21A, RX220
  Refer to -instalign8 for the other devices.

Notes 1. This is when the instruction size is equal to or smaller than the alignment value. If the instruction size is greater than the alignment value, alignment is performed only when the number of exceeding points is two or more.

Notes 2. Alignment is adjusted only for the branch destinations listed above; alignment of the other destinations is not adjusted. For example, when **loop** is selected, alignment of the head of a **loop** is adjusted but alignment is not adjusted at the branch destination of an **if** statement that is used in the loop but does not generate a loop.

- If an object module file or a standard library that has been generated through compilation without using this option is specified for linkage, the warning W0561322 will be output at linkage but program execution will have no problem.

-instalign8

< Compile Options / Object Options >

[Format]

```
-instalign8[={loop|inmostloop}]
```

[Description]

- This option aligns instructions at branch destinations.
- When the **instalign8** option is specified, the instruction at the location address is aligned to the 8-byte boundary.
- Instruction alignment is performed only when the instruction at the specified location exceeds the address which is a multiple of the alignment value (8)*1.
- The following three types of branch destination can be selected by specifying the suboptions of **-instalign4** and **-instalign8***2.
- No specification:Head of function and **case** and **default** labels of **switch** statement
  inmostloop: Head of each inmost loop, head of function, and **case** and **default** labels of **switch** statement
  loop: Head of each loop, head of function, and **case** and **default** labels of **switch** statement
- When these options are selected, the alignment value of the program section is changed from 1 to 4 (for **instalign4**) or 8 (for **instalign8**).
- These options aim to efficiently operate the instruction queues of the RX CPU and improve the speed of program execution by aligning the addresses of branch destination instructions.
- This option is designed for improving the performance for the devices with CPU of the version of RXv2 or later. Refer to -instalign4 for the other devices.

Notes 1. This is when the instruction size is equal to or smaller than the alignment value. If the instruction size is greater than the alignment value, alignment is performed only when the number of exceeding points is two or more.

Notes 2. Alignment is adjusted only for the branch destinations listed above; alignment of the other destinations is not adjusted. For example, when **loop** is selected, alignment of the head of a loop is adjusted but alignment is not adjusted at the branch destination of an **if** statement that is used in the loop but does not generate a loop.

- If an object module file or a standard library that has been generated through compilation without using this option is specified for linkage, the warning W0561322 will be output at linkage but program execution will have no problem.

[Example]

- <C source file>

```
dlong a;
int f1(int num)
{
    return (num+1);
}
void f2(void)
{
    a = 0;
}
void f3(void)
{
}
```

- <Output code>
  [When compiling with -instalign8 specified]
  In the example shown below, the head of each function is aligned so that the instruction does not exceed the 8-byte boundary.
  In 8-byte boundary alignment of instructions, the address will not be changed unless the target instruction exceeds the 8-byte boundary. Therefore, only the address of function f2 is actually aligned.

```
     .SECTION P,CODE,ALIGN=8
     .INSTALIGN 8
_f1:                     ; Function f1, address = 0000H
     ADD     #01H,R1     ; 2 bytes
     RTS                 ; 1 byte
     .INSTALIGN 8
_f2:                     ; Function f2, address =0008H
                         ; Note: Alignment is performed.
                         ; When a 6-byte instruction is placed at
                         ; 0003H, it exceeds the 8-byte boundary.
                         ; Thus, alignment is performed.
     MOV.L   #_a,R4      ; 6 bytes
     MOV.L   #0,[R4]     ; 3 bytes
     RTS                 ; 1 byte
     .INSTALIGN 8
_f3:                     ; Function f3, address = 0012H
      RTS
     .END
```

-noinstalign

< Compile Options / Object Options >

[Format]

```
-noinstalign
```

[Description]

- This option does not aligns instructions at branch destinations.

-nouse_div_inst

< Compile Options / Object Options >

[Format]

```
-nouse_div_inst
```

[Description]

- This option generates code in which no DIV, DIVU, FDIV, or DDIV instructions are used for division and modular division operations in the program.

[Remarks]

- This option calls the equivalent runtime functions instead of DIV, DIVU, FDIV, or DDIV instructions. This may lower code efficiency in terms of required ROM capacity and speed of execution.

-create_unfilled_area [V2.03.00 or later]

< Compile Options / Object Options >

[Format]

```
-create_unfilled_area
```

[Description]

- When a **Motorola S-record file** (**<name>.mot**) or **Hex file** (**<name>.hex**) is output, this option blocks spaces created by **.OFFSET** directives in the assembly language being filled with output data.
- When using this option, specify it when using the **ccrx** or **asrx** command to create an object file (**<name>.obj**) as well as when using the **rlink** command to create a **Motorola S-record file** or **Hex file**.

[Remarks]

- This option is available in V2.03 and later versions of this compiler.
- When this option is used, symbols in the format shown below*1 will be added for each .**OFFSET** directive.

```
__$_<FileName>_<SectionName>_<IDNumber>s__unfilled_area
__$_<FileName>_<SectionName>_<IDNumber>e__unfilled_area
```

Here, the name of the source file, section, and a number in a sequence starting from 1 are entered as **<FileName>**, **<SectionName>**, and **<IDNumber>**, respectively.

Note
*1) Since symbols in this format are reserved, they cannot be directly included in your source code.

-stack_protector/-stack_protector_all [Professional Edition only] [V2.04.00 or later]

< Compile Options / Object Options >

[Format]

```
-stack_protector[=<numeric value>]
-stack_protector_all[=<numeric value>]
```

[Description]

- This option generates a code for detection of stack smashing at the entry and the end of a function. A code for detection of stack smashing consists of instructions executing the three processes shown below.
  (1) A 4-byte area is allocated just before (in the direction towards address 0xFFFFFFFF) the local variable area at the entry to a function, and the value specified by <number> is stored in the allocated area.
  (2) At the end of the function, whether the 4-byte area in which <number> was stored has been rewritten is checked.
  (3) If the 4-byte area has been rewritten in (2), the **__stack_chk_fail** function is called as the stack has been smashed.
- A decimal number from 0 to 4294967295 should be specified in <number>. If the specification of <number> is omitted, the compiler automatically select the number.
- The **__stack_chk_fail** function needs to be defined by the user. It should contain postprocesses for the detected stack smashing.
- Note the following items when defining the **__stack_chk_fail** function.
  - The only possible type of return value is **void** and any formal parameters not allowed.
  - It is prohibited to call the **__stack_chk_fail** function as a normal function.
  - The **__stack_chk_fail function** is not subject to generating a code for detection of stack smashing due to the **-stack_protector** and **-stack_protector_all options** and **#pragma stack_protector**.
  - In a C++ program, add extern "C" to the definition or the declaration for **__stack_chk_fail** function.
  - Prevent returning to the caller (the function where stack smashing was detected) by taking measures such as calling **abort()** in **__stack_chk_fail** function and terminating the program.
  - Do not define the function as **static**.
- If **-stack_protector** is specified, this option generates a code for detection of stack smashing for only functions having a structure, union, or array that exceeds eight bytes as a local variable. If **-stack_protector_all** is specified, this option generates a code for detection of stack smashing for all functions.
- If these options are used simultaneously with **#pragma stack_protector**, the specification by **#pragma stack_protector** becomes valid.
- Even though these options are specified, a code for detection of stack smashing is not generated for a functions for which one of the following **#pragma** directives is specified.
  #pragma inline, **#pragma inline_asm**, **#pragma entry**, **#pragma no_stack_protector**

[Example]

- <C source file>

```
#include <stdio.h>
#include <stdlib.h>

void f1()   // Sample program in which the stack is smashed
{
    volatile char str[10];
    int i;
    for (i = 0; i <= 10; i++){
       str[i] = i; // Stack is smashed when i=10
    }
}

#ifdef __cplusplus
extern "C" {
#endif
void __stack_chk_fail(void)
{
    printf("stack is broken!");
    abort();
}
#ifdef __cplusplus
}
#endif
```

- <Output code>
  When compilation is performed with -stack_protector=0 specified

```
    .glb    _test
    .glb    ___stack_chk_fail
    .glb    _printf
    .glb    _abort
    .SECTION        P,CODE
_test:
    .STACK  _test=20
    MOV.L #00000000H, R14   ; The specified <number> 0 is stored in the stack area.
    PUSH.L R14
    SUB #0CH, R0
    MOV.L #00000000H, R14
    MOV.L #0000000BH, R15
    ADD #02H, R0, R5
L12:    ; parse_bb
    MOV.B R14, [R5+]
    ADD #01H, R14
    SUB #01H, R15
    BNE L12
L13:    ; return
    MOV.L 0CH[R0], R14      ; Data is loaded from the location where <number> was
    CMP #00H, R14           ; stored at the entry to a function and it is compared
                            ; with the specified <number> 0.
    BNE L15                 ; If they do not match, the program branches to L15.
L14:    ; return
    RTSD #10H
L15:    ; return
    BRA ___stack_chk_fail   ; __stack_chk_fail is called.

___stack_chk_fail:
    .STACK  ___stack_chk_fail=8
    SUB #04H, R0
    MOV.L #_L10, R14
    MOV.L R14, [R0]
    BSR _printf
    ADD #04H, R0
    BRA _abort

.SECTION            L,ROMDATA,ALIGN=4
_L10:
    .byte   "stack is broken!"
    .byte   00H
    .END
```

-avoid_cross_boundary_prefetch [V2.07.00 or later]

< Compile Options / Object Options >

[Format]

```
-avoid_cross_boundary_prefetch
```

[Description]

- When both of the conditions given below are satisfied, using this option allows the compiler to expand library functions for handling strings as two units of code that include string manipulation instructions: one for manipulating data at the address where reading of the string starts up to the next 4-byte boundary and the other for manipulating data from that 4-byte boundary up to the last address.
    - Files of source code include calls of library functions for string handling, i.e. memchr(), strlen(), strcpy(), strncpy(), strcmp(), strncmp(), strcat(), or strncat().
    - library=intrinsic has been specified to select the expansion of library functions.

[Remarks]

- The aim of this option is to prevent the reading of data across 4-byte boundaries in prefetching for string manipulation instructions.
- If this option is selected, the code size increases when library functions for string handling, i.e. memchr(), strlen(), strcpy(), strncpy(), strcmp(), strncmp(), strcat(), or strncat(), are compiled with library=intrinsic specified.
- Using this option allows the library generator to expand library functions for string handling, i.e. memchr(), strlen(), strcpy(), strncpy(), strcmp(), strncmp(), strcat(), or strncat(), as two units of code that include string manipulation instructions: one for manipulating data at the address where reading of the string starts up to the next 4-byte boundary and the other for manipulating data from that 4-byte boundary up to the last address.

-insert_nop_with_label [V2.08.00 or later]

< Compile Options / Object Options >

[Format]

```
-insert_nop_with_label=<file>,<line>,<label>
```

- Interpretation when omitted
  A local label and nop instruction are not inserted.

[Description]

- This option inserts a local label and nop instruction at the specified location based on the information for source debugging.
- When this option is specified, the **-debug** option also becomes valid.
- This function is assumed to be used via CS+ or e^{2}studio and should not be used directly by the user.

-control_flow_integrity [Professional Edition only] [V2.08.00 or later]

< Compile Options / Object Options >

[Format]

```
-control_flow_integrity
```

- Interpretation when omitted
  Code for the detection of illegal indirect function calls is not generated.

[Description]

- This option generates code for the detection of illegal indirect function calls.
  When this option is specified, code for the following processing is generated in the C/C++ source program.
    (1) The __control_flow_integrity checking function is called with an indirect calling address as an argument immediately before indirect function calls.
    (2) Within the checking function, the address given as the argument is checked against a list of the addresses of functions (hereafter referred to as the function list) which may be indirectly called. If the list does not include the address, the __control_flow_chk_fail function will be called since this is regarded as an illegal indirect function call.
  The correctness of processing to change the flow of the program, such as through indirect function calls, is referred to as control flow integrity (CFI), and CFI techniques are used to verify this.
- A checking function is defined as follows and provided as library functions.
    void __control_flow_integrity(void *addr);
  Calling the checking function in the same way as normal functions is prohibited.
- The compiler automatically extracts the information on the functions which may be indirectly called from the C/C++ source program. The linker consolidates that information in creating the function list. For the linker to create a function list, the **-cfi** link option must be specified.
  For details, refer to section 2.5.3 Optimizing Linkage Editor (rlink) Options.
- The __control_flow_chk_fail function contains code for the processing which is to be executed when an illegal indirect function call is detected. The user must define this function.
  Note the following when defining the __control_flow_chk_fail function.
    - Specify void as the type of the return value and parameter.
    - Do not define the function as static.
    - Calling the __control_flow_chk_fail function in the same way as a normal function is prohibited.
    - The __control_flow_chk_fail function is not for the creation of code for detecting illegal indirect function calls.
    - In the __control_flow_chk_fail function, note that execution must not be returned to the checking function, for example, by calling abort() to terminate the program.
    - When defining the __control_flow_chk_fail function in a C++ program, add 'extern "C"'.
    - If the **-pic** option is specified at the same time, an error will occur.

[Example]

- <C source code>

```
#include <stdlib.h>

int glb;

void __control_flow_chk_fail(void)
{
  abort();
}

void func1(void) // Added to the function list.
{
  ++glb;
}

void func2(void) // Not added to the function list.
{
  --glb;
}

void (*pf)(void) = func1;

void main(void)
{
  pf(); // Indirect call of the function func1.
  func2();
}
```

- <Output code>
  When **-isa=rxv2 -output=src -control_flow_integrity** is specified for compilation

```
___control_flow_chk_fail:
  .STACK ___control_flow_chk_fail=4
  BRA _abort
_func1:
  .STACK _func1=4
  MOV.L #_glb, R14
  MOV.L [R14], R15
  ADD #01H, R15
  MOV.L R15, [R14]
  RTS
_func2:
  .STACK _func2=4
  MOV.L #_glb, R14
  MOV.L [R14], R15
  SUB #01H, R15
  MOV.L R15, [R14]
  RTS
_main:
  .STACK _main=8
  PUSH.L R6
  MOV.L #_pf, R6
  MOV.L [R6], R1
  BSR ___control_flow_integrity ; Call the checking function.
  MOV.L [R6], R14
  JSR R14 ; Indirect call of the function func1.
  BSR _func2 ; Direct call of the function func2.
  RTSD #04H, R6-R6
  .SECTION D,ROMDATA,ALIGN=4
_pf:
  .lword _func1
  .SECTION B,DATA,ALIGN=4
_glb:
  .blkl 1
  .END
```

List Options

< Compile Options / List Options >

The following list options are available.

- -listfile
- -nolistfile
- -show

-listfile

< Compile Options / List Options >

[Format]

```
-listfile[={<file name>|<path name>}]
```

[Description]

- These options specify whether to output a source list file.
- When the **listfile** option is specified, a source list file is output. <file name> can also be specified.
- An existing folder can also be specified as <path name> instead of <file name>. In such a case, a source list file with the file extension **.lst** and the name of the source file being compiled or assembled is output to the folder selected as <path name>.

[Remarks]

- A linkage list cannot be output by this option. In order to output a linkage list, specify the **list** option of the optimizing linkage editor by using the **lnkcmd** option.
- Information output from the compiler is written to the source list. For the source list file format, refer to 3.1 Assemble List File.
- When you use <path name>, create the folder in advance. If the folder specified as <path name> does not exist, the compiler will assume that <file name> is selected.

-nolistfile

< Compile Options / List Options >

[Format]

```
-nolistfile
```

[Description]

- When the **nolistfile** option is specified, no source list file is output.

-show

< Compile Options / List Options >

[Format]

```
-show=<sub>[,...]
      <sub>: { source | conditionals | definitions | expansions }
```

[Description]

- This option sets the source list file contents.
- The suboptions and specified contents are shown in the following table.

Table 2.5 Suboption Specifications

Suboption	Description
source	Outputs the C/C++ source file.
conditionals	Outputs also the statements for which the specified condition is not satisfied in conditional assembly.
definitions	Outputs the information before **.DEFINE** replacement.
expansions	Outputs the assembler macro expansion statements.

[Remarks]

- This option is valid only when the **listfile** option has been specified.
- Information output from the compiler is written to the source list. For the source list file format, refer to 3.1 Assemble List File.

Optimize Options

< Compile Options / Optimize Options >

The following optimize options are available.

- -optimize
- -goptimize
- -speed
- -size
- -loop
- -inline
- -noinline
- -file_inline
- -case
- -volatile
- -novolatile
- -type_size_access_to_volatile [V3.04.00 or later]
- -const_copy
- -noconst_copy
- -const_div
- -noconst_div
- -library
- -scope
- -noscope
- -schedule
- -noschedule
- -map
- -smap
- -nomap
- -approxdiv
- -enable_register
- -simple_float_conv
- -fpu
- -nofpu
- -dpfpu [V3.01.00 or later]
- -nodpfpu [V3.01.00 or later]
- -tfu [V3.01.00 or later]
- -tfu_version [V3.05.00 or later]
- -nosave_tfu [V3.05.00 or later]
- -alias
- -float_order
- -branch_chaining [V3.03.00 or later]
- -nobranch_chaining [V3.03.00 or later]
- -ip_optimize
- -merge_files
- -whole_program

-optimize

< Compile Options / Optimize Options >

[Format]

```
-optimize = { 0 | 1 | 2 | max }
```

[Description]

- This option specifies the optimization level.
- When **optimize=0** is specified, the compiler does not optimize the program. Accordingly, the debugging information may be output with high precision and source-level debugging is made easier.
- When **optimize=1** is specified, the compiler partially optimizes the program by automatically allocating variables to registers, integrating the function exit blocks, integrating multiple instructions which can be integrated, etc. Accordingly, the code size may become smaller than when compiled with the **optimize=0** specification.
- When **optimize=2** is specified, the compiler performs overall optimization. However, the optimization contents to be performed slightly differ depending on whether the **size** option or **speed** option has been selected.
- When **optimize=max** is specified, the compiler performs optimization as much as possible. For example, the optimization scope is expanded to its maximum extent, and if the **speed** option is specified, loop expansion is possible on a large scale. Though the advantages of optimization can be expected, there may be side effects, such as longer compilation time, and if the **speed** option is specified, significantly increased code size.

[Remarks]

- If the default is not included in the description of an optimize option, this means that the default varies depending on the **optimize** option and **speed** or **size** option specifications. For details on the default, refer to the **speed** or **size** option.

-goptimize

< Compile Options / Optimize Options >

[Format]

```
-goptimize
```

[Description]

- This option generates the additional information for inter-module optimization in the output file.
- At linkage, inter-module optimization is applied to files for which this option has been specified.

-speed

< Compile Options / Optimize Options >

[Format]

```
-speed
```

- [Default]
  Optimization is with emphasis on size.

[Description]

- When the **speed** option is specified, optimization will be performed with emphasis on execution performance.

[Remarks]

- When the **speed** option is specified, the following options are automatically specified based on the **optimize** option specification.
- The processing for optimization in response to the optimization level selected for the **optimize** option includes the fine adjustment of many items other than those that can be specified through compiler options. Code produced with different levels of optimization will differ in the ways set by the compiler options listed in the tables below, but will also not match in other ways.

Optimization item	optimize=0 optimize=1	optimize=2	optimize=max
Loop Expansion	loop=1	loop=2	loop=8
Inline Expansion	noinline	inline=100	inline=250
Converting Constant Division into Multiplication	const_div	const_div	const_div
Scheduling Instructions	noschedule	schedule	schedule
Constant Propagation of const Qualified Variables	noconst_copy	const_copy	const_copy
Dividing Optimizing Ranges	scope	scope	noscope
Optimizing External Variable Accesses	nomap	nomap	map* nomap*
Optimization Considering the Type of the Data Indicated by the Pointer	alias=noansi	alias=noansi	alias=ansi
Optimization by Reducing the Branch Instruction Size	nobranch_chaining	nobranch_chaining	nobranch_chaining

Note The default is **map** when a C/C++ source program has been specified for input and **output=abs** or **output=mot** has been specified for output. For any other case, the default is **nomap**.

-size

< Compile Options / Optimize Options >

[Format]

```
-size
```

- [Default]
  Optimization is with emphasis on size.

[Description]

- When the **size** option is specified, optimization will be performed with emphasis on code size.

[Remarks]

- When the **size** option is specified, the following options are automatically specified based on the **optimize** option specification. Note however that if one of the following options is specified otherwise explicitly, that specified option becomes valid.
- The processing for optimization in response to the optimization level selected for the **optimize** option includes the fine adjustment of many items other than those that can be specified through compiler options. Code produced with different levels of optimization will differ in the ways set by the compiler options listed in the tables below, but will also not match in other ways.

Optimization item	optimize=0 optimize=1	optimize=2	optimize=max
Loop Expansion	loop=1	loop=1	loop=1
Inline Expansion	noinline	noinline	inline=0
Converting Constant Division into Multiplication	noconst_div	noconst_div	noconst_div
Scheduling Instructions	noschedule	schedule	schedule
Constant Propagation of const Qualified Variables	noconst_copy	const_copy	const_copy
Dividing Optimizing Ranges	scope	scope	noscope
Optimizing External Variable Accesses	nomap	nomap	map* nomap*
Optimization Considering the Type of the Data Indicated by the Pointer	alias=noansi	alias=noansi	alias=ansi
Optimization by Reducing the Branch Instruction Size	nobranch_chaining	branch_chaining	branch_chaining

Note The default is **map** when a C/C++ source program has been specified for input and **output=abs** or **output=mot** has been specified for output. For any other case, the default is **nomap**.

-loop

< Compile Options / Optimize Options >

[Format]

```
-loop[=<numeric value>]
```

- [Default]
  The default for this option is **loop=2**.

[Description]

- This option specifies whether to optimize loop expansion.
- When the **loop** option is specified, the compiler expands loop statements (**for**, **while**, and **do-while**).
- The maximum expansion factor can be specified by <numeric value>. An integer from 1 to 32 can be specified for <numeric value>. If no <numeric value> is specified, 2 will be assumed.
- The default for this option is determined based on the **optimize** option and **speed** or **size** option specifications. For details, refer to the **speed** or **size** option.

[Remarks]

- This option is invalid when **optimize=0** or **optimize=1**.

-inline

< Compile Options / Optimize Options >

[Format]

```
-inline[=<numeric value>]
```

- [Default]
  The default for this option is inline=100.

[Description]

- These options specify whether to automatically perform inline expansion of functions.
- A value from 0 to 65535 is specifiable as **<numeric value>**.
- When the **inline** option is specified, the compiler automatically performs inline expansion. However, inline expansion is not performed for the functions specified by **#pragma noinline**. The user is able to use **inline=<numeric value>**, to specify the allowed increase in the function's size due to the use of inline expansion. For example, when **inline=100** is specified, inline expansion will be performed until the function size has increased by 100% (size is doubled).
- The default for this option is determined based on the **optimize** option and **speed** or **size** option specifications. For details, refer to the **speed** or **size** option.

[Remarks]

- Expansion is attempted for the functions with **#pragma inline** specified regardless of the specification of this option. The inline function specifier (C99) is effective only when this option is valid.
- To perform inline expansion for a function for certain, specify **#pragma inline** for the function.
- Even though this option has been selected or an **inline** specifier has been specified for the function, if the compiler judges that the efficiency is degraded by inline expansion, it will not perform it in some cases.

-noinline

< Compile Options / Optimize Options >

[Format]

```
-noinline
```

[Description]

- When the **noinline** option is specified, automatic inline expansion is not performed.

[Remarks]

- Expansion is attempted for the functions with **#pragma inline** specified regardless of the specification of this option. The **inline** function specifier (C99) is effective only when this option is invalid.

-file_inline

< Compile Options / Optimize Options >

[Format]

```
-file_inline=<file name>[,...]
```

[Description]

- This option is not available in V.2.00. Any specification of this option will simply be ignored and will not lead to an error due to compatibility with former versions.

[Remarks]

- For C (C99) source files, **-merge_files** can be used instead of **-file_inline**. Add the file that was used with **-file_inline** (including the file path if **-file_inline_path** was used together with it) as one of the source files to be merged.
- There are some points to be noted regarding **-merge_files**. Refer to [Remarks] of the **-merge_files** option.

-case

< Compile Options / Optimize Options >

[Format]

```
-case={ ifthen | table | auto }
```

- [Default]
  The default for this option is **case=auto**.

[Description]

- This option specifies the expansion method of the **switch** statement.
- When **case=ifthen** is specified, the **switch** statement is expanded using the **if_then** method, which repeats, for each **case** label, comparison between the value of the evaluation expression in the **switch** statement and the **case** label value. If they match, execution jumps to the statement of the **case** label. This method increases the object code size depending on the number of case labels in the **switch** statement.
- When **case=table** is specified, the **switch** statement is expanded by using the table method, where the **case** label jump destinations are stored in a branch table so that a jump to the statement of the **case** label that matches the expression for evaluation in the **switch** statement is made through a single access to the branch table. With this method, the size of the branch table increases with the number of **case** labels in the **switch** statement, but the performance in execution remains the same. The branch table is output to a section for areas holding **switch** statements for branch tables.
- When **case=auto** is specified, the compiler automatically selects the **if_then** method or table method.

[Remarks]

- The branch table created when **case=table** has been specified will be output to section **W** when the **nostuff** option is specified and will be output to section **W**, **W_2**, or **W_1** according to the size of the **switch** statement when the **nostuff** option is not specified.

-volatile

< Compile Options / Optimize Options >

[Format]

```
-volatile
```

[Description]

- When **volatile** is specified, all external variables are handled as if they were **volatile** qualified. Accordingly, the access count and access order for external variables are exactly the same as those written in the C/C++ source file.

[Remarks]

- Debugging tools for RX do not display the volatile declaration added to individual variables by this option.

-novolatile

< Compile Options / Optimize Options >

[Format]

```
-novolatile
```

[Description]

- When **novolatile** is specified, the external variables which are not **volatile** qualified are optimized. Accordingly, the access count and access order for external variables may differ from those written in the C/C++ source file.

-type_size_access_to_volatile [V3.04.00 or later]

< Compile Options / Optimize Options >

[Format]

```
-type_size_access_to_volatile
```

- [Default]
  A variable with **volatile** specified might be accessed in a different size from the size of the variable type.

[Description]

- A variable with **volatile** specified is accessed in the size of the variable type.
- This option also takes effect on the volatile specification by the **-volatile** option.
- Specifying this option more than once has the same effect as specifying it once only. No warning is output in this case.
- For semantic rules for variables with **volatile** specified, this option can only change the access width. Other semantic rules such as the number of accesses cannot be changed.
- If you want to access a variable in the size of its type without this option specified, specify the __evenaccess keyword when declaring the variable.

[Remarks]

- When this option is specified, the access size is guaranteed for 4-byte or smaller scalar types.

[Example]

```
> ccrx a.c -isa=rxv3 -type_size_access_to_volatile
```

-const_copy

< Compile Options / Optimize Options >

[Format]

```
-const_copy
```

- [Default]
  The default for this option is **const_copy** when the **optimize=2** or **optimize=max** option has been specified.

[Description]

- When **const_copy** is specified, constant propagation is performed even for **const** qualified global variables.
- The default for this option is **const_copy** when the **optimize=2** or **optimize=max** option has been specified.

[Remarks]

- **const** qualified variables in a C++ source file cannot be controlled by this option (constant propagation is always performed).

-noconst_copy

< Compile Options / Optimize Options >

[Format]

```
-noconst_copy
```

- [Default]
  The default for this option is **noconst_copy** when the **optimize=1** or **optimize=0** option has been specified.

[Description]

- When **noconst_copy** is specified, constant propagation is disabled for **const** qualified global variables.

[Remarks]

- **const** qualified variables in a C++ source file cannot be controlled by this option (constant propagation is always performed).

-const_div

< Compile Options / Optimize Options >

[Format]

```
-const_div
```

- [Default]
  The default for this option is **const_div** when the **speed** option has been specified.

[Description]

- When **const_div** is specified, calculations for division and remainders of integer constants in the source file are converted into sequences of multiplication or bitwise operation (shift or bitwise AND operations) instructions.

[Remarks]

- Constant multiplication that can be performed through only shift operations and division and residue that can be performed through only bitwise AND operations cannot be controlled by the **const_div** option.

-noconst_div

< Compile Options / Optimize Options >

[Format]

```
-noconst_div
```

- [Default]
  The default for this option is **noconst_div** when the **size** option has been specified.

[Description]

- When **noconst_div** is specified, the corresponding division and remainder instructions are used for calculating division and remainders of integer constants in the source file (except divisions and remainders of unsigned integers by powers of two).

[Remarks]

- Constant multiplication that can be performed through only shift operations and division and residue that can be performed through only bitwise AND operations cannot be controlled by the **noconst_div** option.

-library

< Compile Options / Optimize Options >

[Format]

```
-library = { function | intrinsic }
```

- [Default]
  The default for this option is **library=intrinsic**.

[Description]

- When **-library=function** is specified, all library functions are called by calling subroutines provided by standard library.
- When **-library=intrinsic** is specified, a call for any of the following library functions is replaced with the RX instruction that has the corresponding facility.
  - abs
  - fabsf/fabs[*1]/fabsl[*1]
  - sqrtf[*2]/sqrt[*1*2*3]/sqrtl[*1*2*3]
  - memchr/strlen/strcpy/strncpy/strcmp/strncmp/strcat/strncat

  Notes 1. When **-dbl_size=4** or the **-dpfpu** option is specified

  Notes 2. When the **-isa** option is specified with a value other than rxv1 in combination with the **-fpu** option

  Notes 3. When the **-isa** option is specified with a value other than rxv1 in combination with the **-fpu -dbl_size=4**, or when the **-dpfpu** option is specified

[Remarks]

- The value of variable **errno** is not changed by the call of a library function that was replaced with an instruction.

-scope

< Compile Options / Optimize Options >

[Format]

```
-scope
```

- [Default]
  The default for this option varies depending on the specification of the **optimize** option.
  If the **optimize=max** option is specified, **noscope** is assumed.
  Any specification other than **optimize=max** is assumed to be **scope**.

[Description]

- When the **scope** option is specified, the optimizing ranges of the large-size function are divided into many sections before compilation.
- Use this option at performance tuning because it affects the object performance depending on the program.

-noscope

< Compile Options / Optimize Options >

[Format]

```
-noscope
```

- [Default]
  The default for this option varies depending on the specification of the **optimize** option.
  If the **optimize=max** option is specified, **noscope** is assumed.
  Any specification other than **optimize=max** is assumed to be **scope**.

[Description]

- When the **noscope** option is specified, the optimizing ranges are not divided before compilation. When the optimizing range is expanded, the object performance is generally improved although the compilation time is delayed. However, if registers are not sufficient, the object performance may be lowered. Use this option at performance tuning because it affects the object performance depending on the program.

-schedule

< Compile Options / Optimize Options >

[Format]

```
-schedule
```

- [Default]
  The default for this option is **schedule** when the **optimize=2** or **optimize=max** option has been specified.

[Description]

- When the **schedule** option is specified, instructions are scheduled taking into consideration pipeline processing.

-noschedule

< Compile Options / Optimize Options >

[Format]

```
-noschedule
```

- [Default]
  The default for this option is **noschedule** when the **optimize=1** or **optimize=0** option has been specified.

[Description]

- When the **noschedule** option is specified, instructions are not scheduled. Basically, processing is performed in the same order the instructions have been written in the C/C++ source file.

-map

< Compile Options / Optimize Options >

[Format]

```
-map[= <file name>]
```

- [Default]
  The default for this option is **nomap**. However, the default is **map** if **abs**, **hex**, or **sty** is specified for the **output** option and the **optimize=max** option is specified.

[Description]

- This option optimizes accesses to global variables.
- When the **map** option is specified, a base address is set by using an external symbol-allocation information file created by the optimizing linkage editor, and a code that uses addresses relative to the base address for accesses to global or static variables is generated.
- When accesses to external variables are to be optimized by the map option, how the **map** option is used differs according to the specification of the **output** option.
- [**output=abs**, **output=sty**, or **output=hex** is specified]
  Specify only map. The compiler automatically performs compilation and linkage twice, and a code in which the base address is set based on external symbol allocation information is generated. Note that when **output=abs**, **output=sty**, or **output=hex** is specified simultaneously with **optimize=max**, **map** will be specified implicitly.
- [**output=obj** is specified]
  Compile the source file once without specifying these options, create an external symbol-allocation information file by specifying **map=<file name>** at linkage by the optimizing linkage editor, and then compile the source file again by specifying **map=<file name>** in **ccrx**.

[Example]

- <C source file>

```
long A,B,C;
void func()
{
    A = 1;
    B = 2;
    C = 3;
}
```

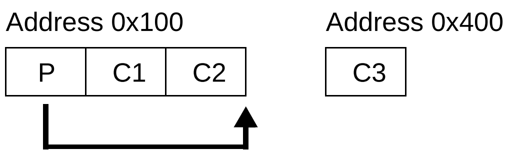
- <Output code>

```
_func:
  MOV.L     #_A,R4     ; Sets the address of A as the base address.
  MOV.L     #1,[R4]
  MOV.L     #2,4[R4]   ; Accesses B using the address of A as the base.
  MOV.L     #3,8[R4]   ; Accesses C using the address of A as the base.
```

[Remarks]

- When the order of the definitions of global variables or static variables has been changed, a new external symbol-allocation information file must be created. If any option other than the **map** option in the previous compilation differs from the one in the current compilation, or if any contents of a function are changed, correct operation is not guaranteed. In such a case, a new external symbol-allocation information file must be created.

- This option is only valid for the compilation of C/C++ source programs. It does not apply to programs that have been compiled with the **output=src** specification or to programs written in assembly language.
- When the **map** option and **smap** option are specified simultaneously, the **map** option is valid.
- When continuous data sections are allocated after a program section, optimization of external variable accesses may be disabled or may not be performed sufficiently. For performing optimization to a maximum extent in a case in which multiple sections are allocated continuously, allocate the program section at the end. An example is shown below.



Note:
P: Program section
C1, C2, C3: Data section

- In the above example, section **P** is allocated from address 0x100, sections **C1** and **C2** are allocated immediately after section **P**, and section **C3** is allocated from address 0x400. Since sections **C1** and **C2** are allocated continuously after section **P**, section **P** should be allocated behind section **C2**. Section **C3** is not involved because it is not allocated continuously.

-smap

< Compile Options / Optimize Options >

[Format]

```
-smap
```

[Description]

- When the **smap** option is specified, a base address is set for global or static variables defined in the file to be compiled, and a code that uses addresses relative to the base address for accesses to those variables is generated.

[Example]

- <C source file>

```
long A,B,C;
void func()
{
    A = 1;
    B = 2;
    C = 3;
}
```

- <Output code>

```
_func:
  MOV.L     #_A,R4     ; Sets the address of A as the base address.
  MOV.L     #1,[R4]
  MOV.L     #2,4[R4]   ; Accesses B using the address of A as the base.
  MOV.L     #3,8[R4]   ; Accesses C using the address of A as the base.
```

[Remarks]

- This option is only valid for the compilation of C/C++ source programs. It does not apply to programs that have been compiled with the **output=src** specification or to programs written in assembly language.
- When the **map** option and **smap** option are specified simultaneously, the **map** option is valid.

-nomap

< Compile Options / Optimize Options >

[Format]

```
-nomap
```

- [Default]
  The default for this option is **nomap**. However, the default is **map** if **abs**, **hex**, or **sty** is specified for the **output** option and the **optimize=max** option is specified.

[Description]

- When the **nomap** option is specified, accesses to external variables are not optimized.

[Example]

- <C source file>

```
long A,B,C;
void func()
{
    A = 1;
    B = 2;
    C = 3;
}
```

- <Output code>

```
_func:
     MOV.L      #_A,R4
     MOV.L      #1,[R4]
     MOV.L      #_B,R4
     MOV.L      #2,[R4]
     MOV.L      #_C,R4
     MOV.L      #3,[R4]
```

-approxdiv

< Compile Options / Optimize Options >

[Format]

```
-approxdiv
```

- [Default]
  When this option is omitted, division of floating-point constants into multiplications of the corresponding reciprocals as constants is not performed.

[Description]

- When there is an expression of (variable ÷ constant), this option generates a code with the expression converted into (variable × reciprocal of constant).

[Remarks]

- When this option is specified, the execution performance of floating-point constant division will be improved. The precision and order of operations may, however, be changed, so take care on this point.

-enable_register

< Compile Options / Optimize Options >

[Format]

```
-enable_register
```

[Description]

- This option is not available in V.2.00. Any specification of this option will simply be ignored and will not lead to an error due to compatibility with former versions.

-simple_float_conv

< Compile Options / Optimize Options >

[Format]

```
-simple_float_conv
```

[Description]

- This option omits part of the type conversion processing for the floating type.
- When this option is selected, the generation code that performs type conversion of the next floating-point number changes.
  - a) Type conversion from 32-bit floating type to unsigned integer type
  - b) Type conversion from unsigned integer type to 32-bit floating type
  - c) Type conversion from integer type to 64-bit floating type via 32-bit floating type

[Example]

- < a) Type conversion from 32-bit floating type to unsigned integer type>

```
unsigned long func1(float f)
{
    return ((unsigned long)f);
}

When this option is not specified:
 _func1:
         FCMP   #4F000000H,R1
         BLT    L12
         FADD   #0CF800000H,R1
 L12:
         FTOI   R1,R1
         RTS
```

- < b) Type conversion from unsigned integer type to 32-bit floating type>

```
float func2(unsigned long u)
{
       return ((float)u);
}

When this option is not specified:
 _func2:
         BTST   #31,R1
         BEQ    L15
         SHLR   #1,R1,R14
         AND    #1,R1
         OR     R14,R1
         ITOF   R1,R1
         FADD   R1,R1
         BRA    L16
 L15:
         ITOF   R1,R1
 L16:
         RTS
```

- < c) Type conversion from integer type to 64-bit floating type via 32-bit floating type>

Does not apply when the dbl_size=8 specification is not valid.

```
double func3(long l)
 {
       return (double)(float)l;
 }
When this option is not specified:
_func3:
        ITOF    R1,R1
        BRA    __COM_CONVfd

When this option is specified:
        BRA    __COM_CONV32sd
```

[Remarks]

- When this option is specified, code performance of the relevant type conversion processing is improved. The conversion result may, however, differ from C/C++ language specifications, so take care on this point.
- This option of c) is invalid when **optimize=0**.

-fpu

< Compile Options / Optimize Options >

[Format]

```
-fpu
```

- [Default]
  The default for this option is **fpu** when the Instruction-code set as the ISA [*1].
  The default for this option is **nofpu** (when RX200 is selected as the target CPU [*2]) or **fpu** (in other cases).

Note
  *1) This means a selection by the **-isa** option or the ISA_RX environment variable.
  *2) This means a selection by the **-cpu** option or the CPU_RX environment variable.

[Description]

- When the **fpu** option is specified, a code using single-precision floating-point processing instructions is generated.

[Remarks]

- For details of the single-precision floating-point processing instructions, refer to the RX Family Software Manual.
- When RX200 is selected as the CPU, an error will occur if **fpu** is specified.

-nofpu

< Compile Options / Optimize Options >

[Format]

```
-nofpu
```

- [Default]
  The default for this option is **fpu** when the Instruction-code set as the ISA [*1].
  The default for this option is **nofpu** (when RX200 is selected as the target CPU [*2]) or **fpu** (in other cases).

Note
  *1) This means a selection by the **-isa** option or the ISA_RX environment variable.
  *2) This means a selection by the **-cpu** option or the CPU_RX environment variable.

[Description]

- When the **nofpu** option is specified, a code not using single-precision floating-point processing instructions is generated.

-dpfpu [V3.01.00 or later]

< Compile Options / Optimize Options >

[Format]

```
-dpfpu
```

- Interpretation when omitted
  nodpfpu is assumed to be specified.

[Description]

- When the **dpfpu** option is specified, a code using double-precision floating-point processing instructions is generated.

[Remarks]

- For details of the double-precision floating-point processing instructions, refer to the RX Family Software Manual.
- When the CPU*1 is selected or RXv1 or RXv2 is selected as ISA*2, an error will occur if **dpfpu** is specified.
- When **nofpu** has been specified, an error will occur if **dpfpu** is specified.

Note
*1) This means a selection by the **cpu** option or the CPU_RX environment variable.
*2) This means a selection by the **isa** option or the ISA_RX environment variable.

-nodpfpu [V3.01.00 or later]

< Compile Options / Optimize Options >

[Format]

```
-nodpfpu
```

- Interpretation when omitted
  nodpfpu is assumed to be specified.

[Description]

- When the **nodpfpu** option is specified, a code not using double-precision floating-point processing instructions is generated.

-tfu [V3.01.00 or later]

< Compile Options / Optimize Options >

[Format]

```
-tfu=intrinsic[,mathlib]
```

[Description]

- When **-tfu=intrinsic** is specified, the following intrinsic functions which use the trigonometric function unit are available.

  - __sincosf
  - __atan2hypotf
  - __init_tfu (for initialization)
  - __sincosfx [V3.05.00 or later]
  - __sinfx [V.05.00 or later]
  - __cosfx [V3.05.00 or later]
  - __atan2fx [V3.05.00 or later]
  - __hypotfx [V3.05.00 or later]
  - __atan2hypotfx [V3.05.00 or later]

  For details on the listed intrinsic functions, refer to section 4.2.6 Intrinsic Functions.

- When **-tfu=intrinsic,mathlib** is specified, the above intrinsic functions are available and calls of relevant mathematics library functions are replaced with code that uses the trigonometric function unit. The following are the mathematics library functions to be replaced.

  - sinf / sin*1 / sinl*1
  - cosf / cos*1 / cosl*1
  - atan2f / atan2*1 / atan2l*1
  - hypotf / hypot*1 / hypotl*1
  - asinf *2 / asin *1*2 / asinl *1*2 [V3.02.00 or later]
  - acosf *2 / acos *1*2 / acosl *1*2 [V3.02.00 or later]
  - atanf / atan *1 / atanl *1 [V3.02.00 or later]
  - tanf / tan *1 / tanl *1 [V3.02.00 or later]

  *1: This is only the case when **-dbl_size=4** is specified.
  *2: This is the case when **-isa=rxv2|rxv3** and **-fpu** are specified.

- [V3.05.00 or later] When **-tfu_version=v2** is specified at the same time, interrupt specifications **tfu** and **no_tfu** specified in #pragma interrupt are available.

[Remarks]

- [Earlier than V3.05.00] Code for operations that use the trigonometric function unit is not reentrant.
- [V3.05.00 or later] Code for operations that use the trigonometric function unit is not reentrant if **-tfu_version=v1** is specified. When **-tfu_version=v2** is specified, the default code is reentrant. However, it is not reentrant if any of the following applies:
    - When **-nosave_tfu** is specified. For details, refer to the description of **-nosave_tfu** in section 2.5.1 Compile Options.
    - When interrupt specification **no_tfu** is specified in #pragma interrupt for any interrupt function that uses the trigonometric function unit. For details, refer to the description of #pragma interrupt in section 4.2.3 #pragma Directive.
- When **-tfu=intrinsic,mathlib** is specified, replacement of the mathematics library functions means that only code from the relevant function calls is replaced and code in the library is not affected. Accordingly, if an indirect call via a pointer is made, the trigonometric function unit will not be used.
- If calls of mathematics library functions are replaced with code that uses the trigonometric function unit, the values of variable **errno** will not be modified.
- Use or non-use of the trigonometric function unit affects the precision of operations.
- When **TFUv1** is used, before using the trigonometric function unit, initialize the unit from the startup program by calling the **__init_tfu()** intrinsic function. If you do not do so, correct operation is not guaranteed.
- When **TFUv2** is used, there is no need to initialize the trigonometric function unit. Calling the **__init_tfu** intrinsic function causes an error.
- Do not specify this option for a device that does not include a trigonometric function unit.

-tfu_version [V3.05.00 or later]

< Compile Options / Optimize Options >

[Format]

```
-tfu_version={v1|v2}
```

- Interpretation when omitted
  If the **-tfu** option is not specified, nothing is assumed to be specified. If the **-tfu** option is specified, v1 is assumed to be specified.

[Description]

- Selects the version of the trigonometric function unit. When **v1** is specified, TFUv1 is selected. When **v2** is specified, TFUv2 is selected.
- If this option is specified with the **-tfu** option not specified, an error (E0511152) occurs.

-nosave_tfu [V3.05.00 or later]

< Compile Options / Optimize Options >

[Format]

```
-nosave_tfu
```

- Interpretation when omitted
  A code for saving and restoring the output of the trigonometric function unit (v2) is generated for all interrupt functions.

[Description]

- A code for saving and restoring the output of the trigonometric function unit (v2) is not generated for any interrupt function. As a result, code for operations that use the trigonometric function unit is not reentrant.
- If **-tfu_version=v2** is not specified, specifying this option causes an error.

[Remarks]

- The code for saving and restoring that is generated when this option is omitted is the same code generated when interrupt specification **tfu** is selected in #pragma interrupt. For the actual code for saving and restoring, refer to the description of #pragma interrupt in section 4.2.3 #pragma Directive.

-alias

< Compile Options / Optimize Options >

[Format]

```
-alias = { noansi | ansi }
```

- [Default]
  The default for this option is alias=noansi.

[Description]

- This option selects whether to perform optimization with consideration for the type of the data indicated by the pointer.
- When alias=ansi is specified, based on the ANSI standard, optimization considering the type of the data indicated by the pointer is performed. Although the performance of object code is generally better than when alias=noansi is specified, the results of execution may differ according to whether alias=ansi or alias=noansi is specified.
- In the same way as in V. 1.00, ANSI-standard based optimization in consideration of the type of data indicated by pointers is not performed when alias=noansi is specified.

[Example]

```
long x;
long n;
void func(short * ps)
{
    n = 1;
    *ps = 2;
    x = n;
}
```

- [When alias=noansi is specified]
  The value of n is reloaded at (A) since it is regarded that there is a possibility of the value of n being rewritten by *ps = 2.

```
_func:
    MOV.L       #_n,R4
    MOV.L       #1,[R4]     ; n = 1;
    MOV.W       #2,[R1]     ; *ps = 2;
    MOV.L       [R4],R5     ; (A) n is reloaded
    MOV.L       #_x,R4
    MOV.L       R5,[R4]
    RTS
```

- [When alias=ansi is specified]
  The value used in assignment at **n = 1** is reused at (B) because it is regarded that the value of **n** will not change at ***ps = 2** since ***ps** and **n** have different types.
  (If the value of **n** is changed by ***ps = 2**, the result is also changed.)

```
_func:
    MOV.L       #_n,R4
    MOV.L       #1,[R4]     ; n = 1;
    MOV.W       #2,[R1]     ; *ps = 2;
    MOV.L       #_x,R4
    MOV.L       #1,[R4]     ; (B) Value used in assignment at n = 1 is reused
    RTS
```

[Remarks]

- When **optimize=0** or **optimize=1** is valid and the **alias** option is specified, the **alias=ansi** specification will be ignored and code will always be generated as if **alias=noansi** has been selected.

-float_order

< Compile Options / Optimize Options >

[Format]

```
-float_order
```

- [Default]
  If this option is omitted, optimization of modification of the operation order in a floating-point expression is not performed.

[Description]

- This option is not available in V.2.00. Any specification of this option will simply be ignored and will not lead to an error due to compatibility with former versions.

-branch_chaining [V3.03.00 or later]

< Compile Options / Optimize Options >

[Format]

```
-branch_chaining
```

[Description]

- This option uses a branch instruction whose code size is small. To use a branch instruction whose code size is small, another branch instruction which shares the same destination may be specified as the branch destination, not a direct branch to the final destination.
- If this option is not selected, the specification for the **optimize**, **speed**, and **size** options is followed.
- For details, see the sections for the **speed** and **size** options.

[Remarks]

- Although this option reduces the code size, it also lowers the execution speed.
- Note that using this optimization without specifying the **-g_line** option may affect the behavior of single-step execution.
- If the **-speed** option is specified, this option is ignored.
- If **-optimize=0** or **1** is specified, this option is ignored.

-nobranch_chaining [V3.03.00 or later]

< Compile Options / Optimize Options >

[Format]

```
-nobranch_chaining
```

[Description]

- This option suppresses the optimization that reduces the branch instruction size.
- If this option is not specified, the specification for the **optimize**, **speed**, and **size** options is followed.
- For details, see the sections for the **speed** and **size** options.

-ip_optimize

< Compile Options / Optimize Options >

[Format]

```
-ip_optimize
```

[Description]

- This option applies global optimization including
  - optimization that utilizes interprocedural alias analysis and
  - propagation of constant parameters and return values.

[Example]

Examples 1.

- <C source code>

```
static int func1(int *a, int *b) {
    *a=0;
    *b=1;
    return *a;
}
int x[2];
int func2() {
    return func1(x, x+1);
}
```

- <Output assembly code without ip_optimize>

```
; -optimize=2 -size
__$func1:
  MOV.L #00000000H, [R1]
  MOV.L #00000001H, [R2]
  MOV.L [R1], R1
  RTS
_func2:
  MOV.L #_x,R1
  ADD #04H, R1, R2
  BRA __$func1
```

- <Output assembly code with ip_optimize>

```
; -optimize=2 -size
__$func1:
  MOV.L #00000000H, [R1]
  MOV.L #00000001H, [R2]
  MOV.L #00000000H, R1
  RTS
_func2:
  MOV.L #_x,R1
  ADD #04H, R1, R2
  BRA __$func1
```

Examples 2.

- <C source code>

```
static int func(int x, int y, int z) {
    return x-y+z;
}
int func2() {
    return func(3,4,5);
}
```

- <Output assembly code without ip_optimize>

```
__$func:
  ADD R3, R1
  SUB R2, R1
  RTS
_func2:
  MOV.L #00000005H, R3
  MOV.L #00000004H, R2
  MOV.L #00000003H, R1
  BRA __$func
```

- <Output assembly code with ip_optimize>

```
__$func:
  MOV.L #00000004H, R1
  RTS
_func2:
  MOV.L #00000005H, R3
  MOV.L #00000004H, R2
  MOV.L #00000003H, R1
  BRA __$func
```

[Remarks]

- Inter-file optimization is also applied when this option is used with **merge_files**.

-merge_files

< Compile Options / Optimize Options >

[Format]

```
-merge_files
```

[Description]

- This option allows the compiler to compile multiple C source files and output the results to a single object file.
- The name of the object file is specified by the output option. If no name is specified, the filename will be that of the first source file plus a filename extension that corresponds to the selected output format.
- If **src** or **obj** is selected as the output format, the compiler also generates blank files that have the names of the other source files with the given filename extension attached.

[Example]

```
ccrx -merge_files -output=obj=files.obj file1.c file2.c file3.c
```

files.obj is the object file. Blank files **file1.obj**, **file2.obj**, and **file3.obj** are also generated.

[Remarks]

- This option is invalid when only one source file is to be compiled or when the **output** option has been used to specify **prep** as the output format.
- Inter-file in-line expansion is applied when this option is used with the **inline** option.
- This option is not available for files to be compiled in C++ or EC++.
- The following restrictions apply to programs that include static functions or static variables.
  - If you wish to use the [Watch] window of the debugger to view a static variable that has the same name as a variable in another file, specify the variable name as well as the filename. The debugger cannot identify the variable without a filename.
  - When two or more files contain static variables with the same name and **rlink** is used to overlay sections to which the files belong, the debugger's facility to display overlay sections taking precedence over other sections is not available.
  - The names of static variables and static functions written in the link map file (.map) are those converted by the compiler (i.e., not original ones).
- Any differences (e.g. type specifier) in declarations of the same variable may lead to an error in compilation.

-whole_program

< Compile Options / Optimize Options >

[Format]

```
-whole_program
```

[Description]

- This option makes the compiler perform optimization on the assumption that all source files have been input.

[Remarks]

- When this option is specified, do not include C++ language source files among the input files.
- When this option is specified, do not specify **-lang=cpp** or **-lang=ecpp**.
- Specifying this option also makes the **ip_optimize** option effective, and if multiple source files are input, the **merge_files** option is also effective.
- When this option is specified, compilation is on the assumption that the conditions listed below are satisfied. Correct operation is not guaranteed otherwise.
    - Values and addresses of **extern** variables defined in the target source files will not be modified or referred to by other files.
    - Functions within the target source file will not be called from within other files, although functions in other files can be called from within the target source files.

[Example]

```
[wp.c]
extern void g(void);
int func(void)
{
    static int a = 0;
    a++;      // (1) Write a value to a.
    g();      // (2) Call g().
    return a; // (3) Call a.
}

[Without whole_program]
The compiler assumes that (2) will change the value of a since function g() may call
function func(), and generates a code to read the value of a in (3).
_func:
        PUSH.L R6
        MOV.L  #__$a$1,R6
        MOV.L  [R6],R14
        ADD    #1,R14
        MOV.L  R14,[R6]     ; (1)
        BSR    _g           ; (2)
        MOV.L  [R6],R1      ; (3)
        RTSD   #4,R6-R6

[With whole_program]
The compiler assumes that function g() will not call function func() and thus (2) will
not change the value of a. As a result, the compiler does not read the value of a in
(3) and instead generates a code to use the value written to a in (1).
_func:
        PUSH.L R6
        MOV.L  #__$a$1,R14
        MOV.L  [R14],R6
        ADD    #1,R6
        MOV.L  R6,[R14]     ; (1)
        BSR    _g           ; (2)
        MOV.L  R6,R1        ; (3)
        RTSD   #4,R6-R6
```

Microcontroller Options

< Compile Options / Microcontroller Options >

The following microcontroller options are available.

- -isa
- -cpu
- -endian
- -round
- -denormalize
- -dbl_size
- -int_to_short
- -signed_char
- -unsigned_char
- -signed_bitfield
- -unsigned_bitfield
- -auto_enum
- -bit_order
- -pack
- -unpack
- -exception
- -noexception
- -rtti
- -fint_register
- -branch
- -base
- -patch
- -pic
- -pid
- -nouse_pid_register
- -save_acc

-isa

< Compile Options / Microcontroller Options >

[Format]

```
-isa={ rxv1 | rxv2 | rxv3 }
```

- [Default]
  The default for this option is determined based on the environment variable **ISA_RX**.

[Description]

- This option specifies an instruction-set architecture for use in generating object files.

[Remarks]

- When neither the **-nofpu** nor **-fpu** option has been selected, specifying this option automatically selects the **-fpu** option.
- Omitting this option when neither the **-cpu** option nor one of the environment variables (CPU_RX or ISA_RX) is specified will lead to an error.
- When the **-cpu** option and this option are specified simultaneously, an error will occur.

-cpu

< Compile Options / Microcontroller Options >

[Format]

```
-cpu={ rx600 | rx200 }
```

- [Default]
  The default for this option is determined based on the environment variable **CPU_RX**.

[Description]

- This option specifies the microcontroller type for the instruction code to be generated.
- When **cpu=rx600** is specified, an instruction code for the RX600 Series is generated.
- When **cpu=rx200** is specified, an instruction code for the RX200 Series is generated.

[Remarks]

- This option is for compatibility with earlier products.
- For upcoming RX-family MCUs, the isa option will be used instead of the cpu option to select an instruction-set architecture. In developing new applications, use the isa option where possible.
- The cpu option can be replaced by the -isa, -fpu and -nofpu option as follows.
- - -cpu=rx600 ==> -isa=rxv1 -fpu
- - -cpu=rx200 ==> -isa=rxv1 -nofpu
- When **cpu=rx200** is specified, the **nofpu** option is automatically selected.
- **cpu=rx200** and the **fpu** option cannot be specified at the same time.
- When **cpu=rx600** is specified while neither the **nofpu** option nor the **fpu** option has been specified, the **fpu** option is automatically selected.
- Omitting the cpu option will lead to an error if neither the -isa option nor one of the environment variables (CPU_RX or ISA_RX) is specified.
- The cpu and isa options cannot be specified at the same time.

-endian

< Compile Options / Microcontroller Options >

[Format]

```
-endian={ big  |  little }
```

- [Default]
  The default for this option is **endian=little**.

[Description]

- When **endian=big** is specified, data bytes are arranged in big endian.
- When **endian=little** is specified, data bytes are arranged in little endian.
- The mode for arranging data bytes can also be specified by the #pragma endian directive. If both this option and a #pragma directive are specified, the #pragma specification takes priority.

-round

< Compile Options / Microcontroller Options >

[Format]

```
-round={ zero  |  nearest }
```

- [Default]
  The default for this option is **round=nearest**.

[Description]

- This option specifies the rounding method for floating-point constant operations.
- When **round=zero** is specified, values are rounded to zero.
- When **round=nearest** is specified, values are rounded to the nearest value.

[Remarks]

- This option does not affect the method of rounding for floating-point operations during program execution.
- The default selection of this option does not affect the selection of the **fpu** and **nofpu** options.

-denormalize

< Compile Options / Microcontroller Options >

[Format]

```
-denormalize={ off  |  on }
```

- [Default]
  The default for this option is **denormalize=off**.

[Description]

- This option specifies the operation when denormalized numbers are used to describe floating-point constants.
- When **denormalize=off** is specified, denormalized numbers are handled as zero.
- When **denormalize=on** is specified, denormalized numbers are handled as they are.

[Remarks]

- This option does not affect the handling of denormalized numbers in floating-point operations during program execution.
- This option is not automatically enabled by the selection of the **fpu** and **nofpu** options.

-dbl_size

< Compile Options / Microcontroller Options >

[Format]

```
-dbl_size={ 4 | 8 }
```

- [Default]
  The default for this option is **dbl_size=8** when the **-dpfpu** option is specified. Otherwise, it is **dbl_size=4**.

[Description]

- This option controls how the **double** type and **long double** type are handled.
  When **dbl_size=4** is specified, they are handled as the single-precision floating-point type.
  When **dbl_size=8** is specified, they are handled as the double-precision floating-point type.

[Remarks]

- When -**dbl_size=4** is selected, among the standard functions, the **mathf.h** and **math.h** functions having the same specifications as each other (e.g., **sqrtf** and **sqrt**) are integrated to configure a standard library. Because of this, phenomena, such as the following example will occur when -**dbl_size=4** is selected. When the RX simulator or emulator traces (single-step execution) the calling of **sqrtf** which is a **mathf.h** header function, it appears as if not **sqrtf** but **sqrt**, which is a **math.h** header function with the same specifications, has been called.

-int_to_short

< Compile Options / Microcontroller Options >

[Format]

```
-int_to_short
```

- [Default]
  Before compilation, the **int** type is not replaced with the **short** type and the **unsigned int** type is not replaced with the **unsigned short** type in the source file.

[Description]

- Before compilation, the **int** type is replaced with the **short** type and the **unsigned int** type is replaced with the **unsigned short** type in the source file.

[Remarks]

- **INT_MAX**, **INT_MIN**, and **UINT_MAX** of **limits.h** are not converted by this option.
- This option is invalid during C++ and EC++ program compilation. If an external name of a C program may be referred to by a C++, EC++ program, message W0523041 will be output for the external name.
- When the **int_to_short** option is specified and a file including a C standard header is compiled as C++ or EC++, the compiler may show the W0523041 message. In this case, simply ignore the message because it does not indicate a problem.
- Data that are shared between C and C++ (EC++) programs must be declared as the **long** or **short** type rather than as the **int** type.
- When an input function having a format such as that of **scanf** in the standard library is called while this option is enabled, be sure to pass the addresses of the variables of the long and unsigned long types as parameters for use in **%d** and **%u** conversion. If the address of the int-type or unsigned-type variables not declared as long is passed, the program might not handle related operations correctly.

-signed_char

< Compile Options / Microcontroller Options >

[Format]

```
-signed_char
```

- [Default]
  When **-signed_char** is omitted, **char** type values are handled as unsigned.

[Description]

- When **-signed_char** is specified, **char** type values are handled as signed.

[Remarks]

- The bit-field members of the **char** type are not controlled by this option; control them using the -**signed_bitfield** and **-unsigned_bitfield** options.

-unsigned_char

< Compile Options / Microcontroller Options >

[Format]

```
-unsigned_char
```

- [Default]
  When **-unsigned_char** is omitted, **char** type values are handled as unsigned.

[Description]

- When **-unsigned_char** is specified, **char** type values are handled as unsigned.

[Remarks]

- The bit-field members of the **char** type are not controlled by this option; control them using the **signed_bitfield** and **unsigned_bitfield** options.

-signed_bitfield

< Compile Options / Microcontroller Options >

[Format]

```
-signed_bitfield
```

- [Default]
  When **signed_bitfield** is omitted, the value is handled as **unsigned**.

[Description]

- When **signed_bitfield** is specified, the value is handled as **signed**.

-unsigned_bitfield

< Compile Options / Microcontroller Options >

[Format]

```
-unsigned_bitfield
```

- [Default]
  When **unsigned_bitfield** is omitted, the value is handled as **unsigned**.

[Description]

- When **unsigned_bitfield** is specified, the value is handled as **unsigned**.

-auto_enum

< Compile Options / Microcontroller Options >

[Format]

```
-auto_enum
```

- [Default]
  The default for this option is to process the enumeration type size as the **signed long** type.

[Description]

- This option processes the enumerated data qualified by **enum** as the minimum data type with which the enumeration value can fit in.
- The possible enumeration values correspond to the data types as shown in the following table.

Table 2.6 Correspondences between Possible Enumeration Values and Data Types

Enumerator		Data Type	
Minimum Value	Maximum Value	When **-unsigned_char** is selected	When **-signed_char** is selected
–128	127	signed char	char
0	255	char	unsigned char
–32768	32767	signed short	signed short
0	65535	unsigned short	unsigned short
Other than above		int *1	int *1

Note
*1) When the **-int_to_short** option has been selected, the signed 4-byte integer type will be selected.

-bit_order

< Compile Options / Microcontroller Options >

[Format]

```
-bit_order = { left | right }
```

- [Default]
  The default for this option is **bit_order=right**.

[Description]

- This option specifies the order of bit-field members.
- When **bit_order=left** is specified, members are allocated from the upper bit.
- When **bit_order=right** is specified, members are allocated from the lower bit.
- The order of bit-field members can also be specified by the **#pragma bit_order** extension. If both this option and a **#pragma** extension are specified, the **#pragma** specification takes priority.

-pack

< Compile Options / Microcontroller Options >

[Format]

```
-pack
```

- [Default]
  The default for this option is unpack. That is, the structure members and class members have boundary alignment values appropriate for their types. The boundary alignment value for structures and classes equals the maximum boundary alignment value for members.

[Description]

- This option sets the boundary alignment value for structure members and class members to 1.
- The boundary alignment value for structure members can also be set to 1 by using the **#pragma pack** directive. If this option and **#pragma** are specified together, the **#pragma** specification takes priority.

[Remarks]

- The boundary alignment values for structure members and class members when these options are specified are shown in the following table.

Table 2.7 Boundary Alignment Values for Structure Members and Class Members When the **pack** Option is Specified

Member Type	**pack** specified	**pack** not specified
char/signed char/unsigned char/bool Note 1	1	1
signed short/unsigned short	1	2
signed int Note 2 /unsigned int Note 2 /signed long/unsigned long/ signed long long/unsigned long long/float/double/ long double/pointer type	1	4

Notes 1. Becomes the same as **unsigned long** when the **-lang=c** option is specified.

Notes 2. Becomes the same as **signed short/unsigned short** when the **int_to_short** option is specified.

-unpack

< Compile Options / Microcontroller Options >

[Format]

```
-unpack
```

- [Default]
  The default for this option is unpack. That is, the structure members and class members have boundary alignment values appropriate for their types. The boundary alignment value for structures and classes equals the maximum boundary alignment value for members.

[Description]

- The structure members and class members have boundary alignment values appropriate for their types.
- The boundary alignment value for structures and classes equals the maximum boundary alignment value for members.

[Remarks]

- The boundary alignment values for structure members and class members when these options are specified are shown in the following table.

Table 2.8 Boundary Alignment Values for Structure Members and Class Members When the **unpack** Option is Specified

Member Type	**unpack** specified	**unpack** not specified
char/signed char/unsigned char/bool [Note 1]	1	1
signed short/unsigned short	2	2
signed int [Note 2]/unsigned int [Note 2]/signed long/unsigned long/ signed long long/unsigned long long/float/double/ long double/pointer type	4	4

Notes 1. Becomes the same as **unsigned long** when the **-lang=c** option is specified.

Notes 2. Becomes the same as **signed short/unsigned short** when the **int_to_short** option is specified.

-exception

< Compile Options / Microcontroller Options >

[Format]

```
-exception
```

- [Default]
  The C++ exceptional handling function (**try**, **catch**, **throw**) is disabled.

[Description]

- The C++ exceptional handling function (**try**, **catch**, **throw**) is enabled.
- The code performance may be lowered.

[Remarks]

- In order to use the C++ exceptional handling function among files, perform the following:
  - Specify **rtti=on**.
  - Do not specify the **noprelink** option in the optimizing linkage editor.
- The **exception** option can be specified only at C++ compilation. The **exception** option is ignored when **lang=cpp** has not been specified and the input file extension is **.c** or **.p**.

-noexception

< Compile Options / Microcontroller Options >

[Format]

```
-noexception
```

- [Default]
  The C++ exceptional handling function (**try**, **catch**, **throw**) is disabled.

[Description]

- The C++ exceptional handling function (**try**, **catch**, **throw**) is disabled.

[Remarks]

- In order to use the C++ exceptional handling function among files, perform the following:
  - Specify **rtti=on**.
  - Do not specify the **noprelink** option in the optimizing linkage editor.
- The **noexception** option can be specified only at C++ compilation. The **noexception** option cannot be specified when **lang=cpp** has not been specified and the input file extension is **.c** or **.p**. If specified, an error will occur.

-rtti

< Compile Options / Microcontroller Options >

[Format]

```
-rtti={ on | off }
```

- [Default]
  The default for this option is **rtti=off**.

[Description]

- This option enables or disables runtime type information.
- When **rtti=on** is specified, **dynamic_cast** and **typeid** are enabled.
- When **rtti=off** is specified, **dynamic_cast** and **typeid** are disabled.

[Remarks]

- Do not define relocatable files (**.obj**) that were created by this option in a library, and do not output files in the relocatable format (**.rel**) through the optimizing linkage editor. A symbol double definition error or symbol undefined error may occur.
- **rtti=on** can be specified only at C++ compilation. **rtti=on** is ignored when **lang=cpp** has not been specified and the input file extension is **.c** or **.p**.

-fint_register

< Compile Options / Microcontroller Options >

[Format]

```
-fint_register = {0 | 1 | 2 | 3 | 4 }
```

- [Default]
  The default for this option is **fint_register=0**.

[Description]

- This option specifies the general registers which are to be used only in fast interrupt functions (functions that have the fast interrupt setting (**fint**) in their interrupt specification defined by **#pragma interrupt**). The specified registers cannot be used in functions other than the fast interrupt functions. Since the general registers specified by this option can be used without being saved or restored in fast interrupt functions, the execution speed of fast interrupt functions will most likely be improved. Then again, since the number of usable general registers in other functions is reduced, the efficiency of register allocation in the entire program is degraded.
- The options correspond to the registers as shown in the following table.

Table 2.9 Correspondences between Options and Registers

Option	Registers for Fast Interrupts Only
fint_register=0	None
fint_register=1	R13
fint_register=2	R12, R13
fint_register=3	R11, R12, R13
fint_register=4	R10, R11, R12, R13

[Remarks]

- Correct operation is not guaranteed when a register specified by this option is used in a function other than the fast interrupt functions. If a register specified by this option has been specified by the **base** option, an error will occur.

-branch

< Compile Options / Microcontroller Options >

[Format]

```
-branch = { 16 | 24 | 32 }
```

- [Default]
  The default for this option is **branch=24**.

[Description]

- This option specifies the branch width.
- When **branch=16** is specified, the program is compiled with a branch width within 16 bits.
- When **branch=24** is specified, the program is compiled with a branch width within 24 bits.
- When **branch=32** is specified, the branch width is not specified.

-base

< Compile Options / Microcontroller Options >

[Format]

```
-base = { rom=<register>
        | ram=<register>
        | <address value> = <register>}
                            <register>:= {R8 to R13}
```

[Description]

- This option specifies the general register used as a fixed base address throughout the program.
- When **base=rom=<register A>** is specified, accesses to **const** variables are performed relative to the specified register A. Note, however, that the difference between the address closest to 0 and the address closest to 0xFFFFFFFF is within the range from 64 Kbytes to 256 Kbytes*1 in the constant area section.
  The constant area section includes the sections (before renamed) shown below;
  C_1, C_2, C, C_8, C$VECT, C$INIT, C$VTBL, W, W_1, W_2, L
- When **base=ram=<register B>** is specified, accesses to initialized variables and uninitialized variables are performed relative to the specified register B. Note, however, that the difference between the address closest to 0 and the address closest to 0xFFFFFFFF is within the range from 64 Kbytes to 256 Kbytes*1 in the RAM data area section.
  The RAM data area section includes the sections (before renamed) shown below;
  D_1, D_2, D, D_8, B_1, B_2, B, B_8
- When **<address value>=<register C>** is specified, accesses to an area within 64Kbytes to 256 bytes from the address value, among the areas whose addresses are already determined at the time of compilation, are performed relative to the specified register C.

Note
*1) This value is in the range from 64 to 256 Kbytes and depends on the total size of variables to be accessed.

[Remarks]

- The same register cannot be specified for different areas.
- Only a single register can be specified for each area. If a register specified by the **fint_register** option is specified by this option, an error will occur.
- When the **pid** option is selected, **base=rom=<register>** cannot be selected. If selected, message W0523039 is output as a warning and the selection of **base=rom=<register>** is disabled.

-patch

< Compile Options / Microcontroller Options >

[Format]

```
-patch = { rx610 }
```

[Description]

- This option is used to avoid a problem specific to the CPU type.
- When **-patch=rx610** is specified, the **MVTIPL** instruction which causes a problem in the RX610 Group is not used in the generated code. Unless **-patch=rx610** is specified, the code generated in response to the call by the intrinsic function **set_ipl** will contain the **MVTIPL** instruction.

-pic

< Compile Options / Microcontroller Options >

[Format]

```
-pic
```

- [Default]
  This option does not generate code with the program section as PIC (position independent code).

[Description]

- This option generates code with the program section as PIC (position independent code).
- In PIC, all function calls are performed with BSR or BRA instructions. When acquiring the address of a function, a relative address from the PC should be used. This allows PIC to be located at a desired address after linkage.

[Example]

- Calling a function (only for **branch=32**)

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

```
[Without -pic]
_func:
            MOV.L       #_sub,R14
            JMP         R14
[With -pic]
_func:
            MOV.L       #_sub-L11,R14
L11:
            BRA         R14
```

- Acquiring a function address

```
void func1(void);
void (*f_ptr)(void);
void func2(void)
{
        f_ptr = func1;
}
```

```
[Without -pic]
_func2:
            MOV.L       #_f_ptr,R4
            MOV.L       #_func1,[R4]
            RTS
[With -pic]
_func2:
         MOV.L       #_f_ptr,R4
L11:
         MVFC        PC,R14
         ADD         #_func1-L11,R14
         MOV.L       R14,[R4]
         RTS
```

[Remarks]

- In C++ or EC++ compilation, the **pic** option cannot be selected. If selected, message W0511171 is output as a warning and the selection of the **pic** option is disabled.
- The address of a function which is PIC should not be used in the initialization expression used for static initialization. If used, error E0523026 will occur.
- <Example of using a PIC address for static initialization>

```
void pic_func1(void), pic_func2(int), pic_func3(int); /* Becomes PIC */
void (*fptr1_for_pic) = pic_func1; /* Uses PIC address in static initialization:
Error */
struct PIC_funcs{ int code; void (*fptr)(int); };
struct PIC_funcs pic_funcs[] = {
    { 2, pic_func2 },             /* Uses PIC address in static initialization: Error
*/
    { 3, pic_func3 },             /* Uses PIC address in static initialization: Error
*/
};
```

- When creating a code for startup of the application program using the PIC function, refer to the Application Startup section of the STARTUP chapter.
- For the PIC function, also refer to the Usage of PIC/PID Function of the STARTUP section.

-pid

< Compile Options / Microcontroller Options >

[Format]

```
-pid[={ 16 | 32 }]
```

- [Default]
  The constant area sections **C_8**, **C**, **C_2**, and **C_1**, the literal section **L**, and the **switch** statement branch table sections **W**, **W_2**, and **W_1** are not handled as PID (position independent data).

[Description]

- The constant area sections **C_8**, **C**, **C_2**, and **C_1**, the literal section **L**, and the **switch** statement branch table sections **W**, **W_2**, and **W_1** are handled as PID (position independent data).
- PID can be accessed through a relative address from the PID register. This allows PID to be located at a desired address after linkage.
- A single general register is used to implement the PID function.
- <PID register>
  - Based on the rules in the following table, one register from among R9 to R13 is selected according to the specification of the **fint_register** option. If the **fint_register** option is not specified, R13 is selected.

Table 2.10 Correspondences between fint_register Options and PID Registers

fint_register Option	PID Register
No **fint_register** specification	R13
fint_register = 0	
fint_register = 1	R12
fint_register = 2	R11
fint_register = 3	R10
fint_register = 4	R9

  - The PID register can be used only for the purpose of PID access.
- <Parameters>
  - The parameter selects the maximum bit width of the offset when accessing the constant area section from the PID register as 16 bits or 32 bits.
  - The default for this option when the offset width is omitted is **pid=16**. When **pid=16** is specified, the size of the constant area section that can be accessed by the PID register is limited to 64 Kbytes to 256 Kbytes (varies depending on the access width). When **pid=32** is specified, there is no limitation of the size of the constant area section that can be accessed by the PID register, but the size of the code accessing PID is increased.
  - Note that when **pid=32** and the map option with valid external symbol-allocation information are specified at the same time, the allocation information causes code the same as if **pid=16** was specified to be generated if access by the PID register is possible.

[Examples]

- Accessing an externally referenced symbol that is **const** qualified

```
extern const int pid;
int work;
void func1()
{
        work = pid;
}
```

```
[Without -pid]
_func1:
            MOV.L       #_pid,R4
            MOV.L       [R4],R5
            MOV.L       #_work,R4
            MOV.L       R5,[R4]
            RTS
[With -pid=16] (only when the PID register is R13)
_func1:
            MOV.L       _pid-__PID_TOP:16[R13],R5
            MOV.L       #_work,R4
            MOV.L       R5,[R4]
            RTS
            .glb        __PID_TOP
[With -pid=32] (only when the PID register is R13)
_func1:
            ADD         #(_pid-__PID_TOP),R13,R6
            MOV.L       [R6],R5
            MOV.L       #_work,R4
            MOV.L       R5,[R4]
            RTS
            .glb        __PID_TOP
```

- Acquiring the address of an externally defined symbol that is **const** qualified

```
extern const int pid = 1000;
const int *ptr;
void func2()
{
        ptr = &pid;
}
```

```
[Without -pid]
_func2:
            MOV.L       #_ptr,R4
            MOV.L       #_pid,[R4]
            RTS
[With -pid] (only when the PID register is R13)
_func2:
            ADD         #(_pid-__PID_TOP),R13,R5
            MOV.L       #_ptr,R4
            MOV.L       R5,[R4]
            RTS
            .glb        __PID_TOP
```

[Remarks]

- The address of an area which is PID should not be used in the initialization expression used for static initialization. If used, error E0523027 will occur.

- <Example of using a PID address for static initialization>

```
extern const int pid_data1;          /* Becomes PID */
const int *ptr1_for_pid = &pid_data1;/* Uses PID address in static initialization: 
Error */
const int pid_data4[] = {1,2,3,4};   /* Becomes PID */
const int *ptr2_for_pid = pid_data4; /* Uses PID address in static initialization: 
Error */
```

- When creating a code for startup of the application program using the PID function, refer to Application Startup, instead of Startup.
- When the **pid** option is selected, the same external variables in different files all have to be **const** qualified. This is because the **pid** option is used to specify **const** qualified variables as PID. The **pid** option (PID function) should not be used when there may be an external variable that is not **const** qualified.
- If the **map=<file name>** option is enabled while the **pid** option is selected, warning W0530809 may be output when there is an externally referenced variable that is not **const** qualified but used in different files as the same external variable. In the case, the displayed variable is handled as PID.
- In C++ or EC++ compilation, the **pid** option cannot be selected. If selected, message W0511171 is output as a warning and the selection of the **pid** option is disabled.
- When the **pid** option is selected, **base=rom=<register>** cannot be selected. If selected, message W0551149 is output as a warning and the selection of **base=rom=<register>** is disabled.
- If a PID register selected by the **pid** option is also specified by the **base** option, warning W0511149 will occur.
- If the **pid** option and **nouse_pid_register** option are selected simultaneously, error E0511150 will occur.
- For details of the application and PID function, refer to Usage of PIC/PID Function.

-nouse_pid_register

< Compile Options / Microcontroller Options >

[Format]

```
-nouse_pid_register
```

[Description]

- When this option is specified, the generated code does not use the PID register.
- Selection of the PID register according to the settings of the **fint_register** option is based on the same rule as for the **pid** option.
- A master program called by an application program in which the PID function is enabled needs to be compiled with this option. At this time, if the **fint_register** option is selected in the application program, the same parameter **fint_register** should also be selected in the master program.

[Remarks]

- If the **nouse_pid_register** option and **pid** option are selected simultaneously, error E0511150 will occur.
- A register selected as the PID register also being specified for the **base** option leads to warning W0511149.
- For details of the PID function, refer to Usage of PIC/PID Function.

-save_acc

< Compile Options / Microcontroller Options >

[Format]

```
-save_acc
```

- [Default]
  When this option is omitted, it does not generate the saved and restored code of the accumulator (**ACC,ACC0,ACC1**) for interrupt functions.

[Description]

- This option generates the saved and restored code of the accumulator (**ACC,ACC0,ACC1**) for interrupt functions.
- The code for saving and restoring the **ACC** is generated when RXv1 is selected for ISA[*1] or the microcomputer type is selected by the CPU[*2].
- The code for saving and restoring **ACC0** and **ACC1** is generated when a value other than RXv1 is selected for ISA[*1].

Note
*1) This means a selection by the **-isa** option or the ISA_RX environment variable.
*2) This means a selection by the **-cpu** option or the CPU_RX environment variable.

[Remarks]

- The generated code for saving and restoring is the same code generated when interrupt specification "acc" is selected in #pragma interrupt. For the actual code for saving and restoring, refer to the description of #pragma interrupt in section 4.2.3 #pragma Directive.
- Since the value of the accumulator is retained even at interrupt occurrence if this option has been specified, a code using DSP instructions, such as the MACLO instruction, may be generated for C/C++ arithmetic expressions.

Assemble and Linkage Options

< Compile Options / Assemble and Linkage Options >

The following assemble and linkage options are available.

- -asmcmd
- -lnkcmd
- -asmopt
- -lnkopt

-asmcmd

< Compile Options / Assemble and Linkage Options >

[Format]

```
-asmcmd=<file name>
```

[Description]

- This option specifies the assembler options to pass to **asrx** with a subcommand file.

[Example]

```
ccrx -isa=rxv1 -asmcmd=file.sub sample.c
```

The above description has the same meaning as the following two command lines:

```
ccrx -isa=rxv1 -output=src sample.c
asrx -isa=rxv1 -subcommand=file.sub sample.src
```

[Remarks]

- If this option is specified for more than one time, all specified subcommand files are valid.

-lnkcmd

< Compile Options / Assemble and Linkage Options >

[Format]

```
-lnkcmd=<file name>
```

[Description]

- This option specifies the linkage options to pass to **rlink** with a subcommand file.

[Example]

```
ccrx -isa=rxv1 -output=abs=tp.abs -lnkcmd=file.sub tp1.c tp2.c
```

The above description has the same meaning as the following three command lines:

```
ccrx -isa=rxv1 -output=src tp1.c tp2.c]
asrx -isa=rxv1 tp1.src tp2.src
rlink -subcommand=file.sub -form=abs -output=tp tp1.obj tp2.obj
```

[Remarks]

- If this option is specified for more than one time, all specified subcommand files are valid.
- Refer to the **-subcommand** option of the optimizing linkage editor for the contents of the subcommand file passed to the **-lnkcmd** option.

-asmopt

< Compile Options / Assemble and Linkage Options >

[Format]

```
-asmopt=["]<assembler option>["]
```

[Description]

- This option specifies the assembler options to pass to **asrx** with a string.
- Multiple options can be specified by enclosing them with double-quote marks (").

[Example]

```
ccrx -isa=rxv1 -asmopt="-chkpm" sample.c
```

The above description has the same meaning as the following two command lines:

```
ccrx -isa=rxv1 -output=src sample.c
asrx -isa=rxv1 -chkpm sample.src
```

[Remarks]

- If this option is specified for more than one time, all specified assembler options are valid.

-lnkopt

< Compile Options / Assemble and Linkage Options >

[Format]

```
-lnkopt=["]<linkage option>["]
```

[Description]

- This option specifies the linkage options to pass to **rlink** with a string.
- Multiple options can be specified by enclosing them with double-quote marks (").

[Example]

```
ccrx -isa=rxv1 -output=abs=tp.abs -lnkopt="-start=P,C,D/100,B/8000" tp1.c tp2.c
```

The above description has the same meaning as the following three command lines:

```
ccrx -isa=rxv1 -output=src tp1.c tp2.c
asrx -isa=rxv1 tp1.src tp2.src
rlink -start=P,C,D/100,B/8000 -form=abs -output=tp tp1.obj tp2.obj
```

[Remarks]

- If this option is specified for more than one time, all specified linkage options are valid.
- A single **-lnkopt** option can only take a single linkage option. To pass multiple linkage options, specify **-lnkopt** options as many times as the number of linkage options you require.

Other Options

< Compile Options / Other Options >

The following other options are available.

- -logo
- -nologo
- -euc
- -sjis
- -latin1
- -utf8
- -big5
- -gb2312
- -outcode
- -subcommand

-logo

< Compile Options / Other Options >

[Format]

```
-logo
```

- [Default]
  The copyright notice is output.

[Description]

- The copyright notice is output.

-nologo

< Compile Options / Other Options >

[Format]

```
-nologo
```

- [Default]
  The copyright notice is output.

[Description]

- When the **nologo** option is specified, output of the copyright notice is disabled.

-euc

< Compile Options / Other Options >

[Format]

```
-euc
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **EUC** code.

-sjis

< Compile Options / Other Options >

[Format]

```
-sjis
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

-latin1

< Compile Options / Other Options >

[Format]

```
-latin1
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **ISO-Latin1** code.

-utf8

< Compile Options / Other Options >

[Format]

```
-utf8
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **UTF-8** code.

-big5

< Compile Options / Other Options >

[Format]

```
-big5
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **Big5** code.

[Remarks]

- When **big5** is specified, the same character coding must be selected for the **outcode** option.

-gb2312

< Compile Options / Other Options >

[Format]

```
-gb2312
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **GB2312** code.

[Remarks]

- When **gb2312** is specified, the same character coding must be selected for the **outcode** option.

-outcode

< Compile Options / Other Options >

[Format]

```
-outcode = { euc | sjis | utf8 | big5 | gb2312 }
```

- [Default]
  The default for this option is **outcode=sjis**.

[Description]

- This option specifies the character code to output characters in strings and character constants.
- The options correspond to the character codes as shown in the following table.

Table 2.11 Correspondences between Options and Character Codes (outcode)

Option	Character Code
euc	EUC code
sjis	SJIS code
utf8	UTF-8 code
big5	Big5 code
gb2312	GB2312 code

[Remarks]

- When **outcode=big5** or **outcode=gb2312**, the **big5** or **gb2312** option must also be specified.

-subcommand

< Compile Options / Other Options >

[Format]

```
-subcommand=<subcommand file name>
```

[Description]

- When the **subcommand** option is specified, the compiler options specified in a subcommand file are used at compiler startup. Specify options in a subcommand file in the same format as in the command line.

[Remarks]

- If this option is specified for more than one time, all specified subcommand files are valid.

2.5.2 Assembler Command Options

Classification	Option	Description
Source Options	-include	Specifies the names of folders that hold include files.
	-define	Specifies macro definitions.
	-chkpm	Checks whether a privileged instruction is written.
	-chkfpu	Checks whether a single-precision floating-point processing instruction is written.
	-chkdsp	Checks whether a DSP instruction is written.
	-chkdpfpu [V3.01.00 or later]	Checks whether a double-precision floating-point processing instruction is written.
Object Options	-output	Specifies the relocatable file name.
	-debug	Debugging information is output to the object files.
	-nodebug	Debugging information is not output to the object files.
	-goptimize	Outputs additional information for inter-module optimization.
	-fpu	Enables writing of a single-precision floating-point processing instruction.
	-nofpu	Generates an error when a single-precision floating-point processing instruction is written.
	-dpfpu [V3.01.00 or later]	Enables writing of a double-precision floating-point processing instruction.
	-nodpfpu [V3.01.00 or later]	Generates an error when a double-precision floating-point processing instruction is written.
	-bank [V3.01.00 or later]	Enables writing of the SAVE and RSTR instructions dedicated to the register bank save function.
	-nobank [V3.01.00 or later]	Generates an error when the SAVE or RSTR instruction dedicated to the register bank save function is written.
	-create_unfilled_area [V2.03.00 or later]	[To be supported by V2.03 and later versions] Makes spaces created by .OFFSET unfilled.
List Options	-listfile	An assembler list file is output.
	-nolistfile	An assembler list file is not output.
	-show	Specifies the contents of the source list file.

Classification	Option	Description
Microcontroller Options	-isa	Selects the instruction-set architecture.
	-cpu	Selects the microcontroller type.
	-endian	Selects the endian type.
	-fint_register	Selects a general register for exclusive use with the fast interrupt function.
	-base	Specifies the base registers for ROM and RAM.
	-patch	Selects avoidance or non-avoidance of a problem specific to the CPU type.
	-pic	Enables the PIC function.
	-pid	Enables the PID function.
	-nouse_pid_register	The PID register is not used in code generation.
Other Options	-logo	Selects the output of copyright information.
	-nologo	Selects the non-output of copyright information.
	-subcommand	Specifies a file for including command options.
	-euc	The character codes of input programs are interpreted as EUC codes.
	-sjis	The character codes of input programs are interpreted as SJIS codes.
	-latin1	The character codes of input programs are interpreted as ISO-Latin1 codes.
	-big5	The character codes of input programs are interpreted as BIG5 codes.
	-gb2312	The character codes of input programs are interpreted as GB2312 codes.
	-utf8 [V2.04.00 or later]	The character codes of input programs are interpreted as UTF-8 codes.

Source Options

< Assembler Command Options / Source Options >

The following source options are available.

- -include
- -define
- -chkpm
- -chkfpu
- -chkdsp
- -chkdpfpu [V3.01.00 or later]

-include

< Assembler Command Options / Source Options >

[Format]

```
-include=<path name>[,...]
```

- [Default]
  The include file is searched for in the order of the current folder and the folders specified by environment variable **INC_RXA**.

[Description]

- This option specifies the name of the path to the folder that stores the include file.
- Multiple path names can be specified by separating them with a comma (,).
- The include file is searched for in the order of the current folder, the folders specified by the **include** option, and the folders specified by environment variable **INC_RXA**.

[Example]

- Folders **c:\usr\inc** and **c:\usr\rxc** are searched for the include file.

```
asrx -include=c:\usr\inc,c:\usr\rxc test.src
```

-define

< Assembler Command Options / Source Options >

[Format]

```
-define=<sub>[,...]
        <sub>: <macro name> = <string>
```

[Description]

- This option replaces the macro name with the specified string.
  (This provides the same function as writing the **.DEFINE** directive at the beginning of the source file.)

[Remarks]

- **.DEFINE** takes priority over the **define** option if both are specified.

-chkpm

< Assembler Command Options / Source Options >

[Format]

```
-chkpm
```

[Description]

- This option outputs warning W0551011 when a privileged instruction is written in the source file.

[Remarks]

- For details of the privileged instructions, refer to the RX Family Software Manual.

-chkfpu

< Assembler Command Options / Source Options >

[Format]

```
-chkfpu
```

[Description]

- This option outputs warning W0551012 when a single-precision floating-point processing instruction is written in the source file.

[Remarks]

- For details of the single-precision floating-point processing instructions, refer to the RX Family Software Manual.

-chkdsp

< Assembler Command Options / Source Options >

[Format]

```
-chkdsp
```

[Description]

- This option outputs warning W0551013 when a DSP instruction is written in the source file.

[Remarks]

- For details of the DSP instructions, refer to the RX Family Software Manual.

-chkdpfpu [V3.01.00 or later]

< Assembler Command Options / Source Options >

[Format]

```
-chkdpfpu
```

[Description]

- This option outputs warning W0551017 when a double-precision floating-point processing instruction is written in the source file.

[Remarks]

- For details of the double-precision floating-point processing instructions, refer to the RX Family Software Manual.

Object Options

< Assembler Command Options / Object Options >

The following object options are available.

- -output
- -debug
- -nodebug
- -goptimize
- -fpu
- -nofpu
- -dpfpu [V3.01.00 or later]
- -nodpfpu [V3.01.00 or later]
- -bank [V3.01.00 or later]
- -nobank [V3.01.00 or later]
- -create_unfilled_area [V2.03.00 or later]

-output

< Assembler Command Options / Object Options >

[Format]

```
-output=<output file name>
```

- [Default]
  This option outputs a relocatable file having the same name as that of the source file with extension **.obj**.

[Description]

- When the specified output file name does not have an extension, the file name appended with extension **.obj** is used for the output relocatable file name. When it has an extension, the extension is replaced with **.obj**.

-debug

< Assembler Command Options / Object Options >

[Format]

```
-debug
```

- [Default]
  If this option is not specified, no debugging information is output to the relocatable file.

[Description]

- When the **debug** option is specified, debugging information is output to the relocatable file.

-nodebug

< Assembler Command Options / Object Options >

[Format]

```
-nodebug
```

- [Default]
  If this option is not specified, no debugging information is output to the relocatable file.

[Description]

- When the **nodebug** option is specified, no debugging information is output to the relocatable file.

-goptimize

< Assembler Command Options / Object Options >

[Format]

```
-goptimize
```

- [Default]
  If this option is not specified, additional information for the inter-module optimization is not output.

[Description]

- This option outputs the additional information for the inter-module optimization.
- At linkage, inter-module optimization is applied to the file specified with this option.

-fpu

< Assembler Command Options / Object Options >

[Format]

```
-fpu
```

- [Default]
  The default for this option is **fpu** when the Instruction-code set as the ISA *1.
  The default for this option is **nofpu** (when RX200 is selected as the target CPU *2) or **fpu** (in other cases).

Note
  *1) This means a selection by the **-isa** option or the ISA_RX environment variable.
  *2) This means a selection by the **-cpu** option or the CPU_RX environment variable.

[Description]

- This option enables writing of a single-precision floating-point processing instruction.

[Remarks]

- Specifying **fpu** will lead to an error when the RX200 is selected as the target CPU.
- For details of the single-precision floating-point processing instructions, refer to the RX Family Software Manual.

-nofpu

< Assembler Command Options / Object Options >

[Format]

```
-nofpu
```

- [Default]
  The default for this option is **fpu** when the Instruction-code set as the ISA [*1].
  The default for this option is **nofpu** (when RX200 is selected as the target CPU [*2]) or **fpu** (in other cases).

Note
  *1) This means a selection by the **-isa** option or the ISA_RX environment variable.
  *2) This means a selection by the **-cpu** option or the CPU_RX environment variable.

[Description]

- This option generates an error when a single-precision floating-point processing instruction is written.

[Remarks]

- For details of the single-precision floating-point processing instructions, refer to the RX Family Software Manual.
- When this option is specified, a code including single-precision floating-point processing instructions or control register FPSW will cause an error.

-dpfpu [V3.01.00 or later]

< Assembler Command Options / Object Options >

[Format]

```
-dpfpu
```

- Interpretation when omitted
  nodpfpu is assumed to be specified.

[Description]

- This option enables writing of a double-precision floating-point processing instruction.

[Remarks]

- When the CPU*1 is selected or RXv1 or RXv2 is selected as ISA*2, an error will occur if **dpfpu** is specified.
- For details of the double-precision floating-point processing instructions, refer to the RX Family Software Manual.
- When **nofpu** has been specified, an error will occur if **dpfpu** is specified.

Note
*1) This means a selection by the **cpu** option or the CPU_RX environment variable.
*2) This means a selection by the **isa** option or the ISA_RX environment variable.

-nodpfpu [V3.01.00 or later]

< Assembler Command Options / Object Options >

[Format]

```
-nodpfpu
```

- Interpretation when omitted
  nodpfpu is assumed to be specified.

[Description]

- This option generates an error when a double-precision floating-point processing instruction is written.

[Remarks]

- For details of the double-precision floating-point processing instructions, refer to the RX Family Software Manual.
- When this option is specified, a code including double-precision floating-point processing instructions, double-precision floating-point data registers (DR0 to DR15), or double-precision floating-point control registers (DPSW, DMCR, DECNT, and DEPC) will cause an error.

-bank [V3.01.00 or later]

< Assembler Command Options / Object Options >

[Format]

```
-bank
```

- Interpretation when omitted
  When the CPU[*1] is selected or RXv1 or RXv2 is selected as ISA[*2], **-nobank** is assumed to be specified.
  In other cases, **-bank** is assumed to be specified.

  Note
  *1) This means a selection by the **cpu** option or the CPU_RX environment variable.
  *2) This means a selection by the **isa** option or the ISA_RX environment variable.

[Description]

- This option enables writing of the SAVE and RSTR instructions dedicated to the register bank save function.

-nobank [V3.01.00 or later]

< Assembler Command Options / Object Options >

[Format]

```
-nobank
```

- Interpretation when omitted
  When the CPU[*1] is selected or RXv1 or RXv2 is selected as ISA[*2], **-nobank** is assumed to be specified.
  In other cases, **-bank** is assumed to be specified.

  Note
  *1) This means a selection by the **cpu** option or the CPU_RX environment variable.
  *2) This means a selection by the **isa** option or the ISA_RX environment variable.

[Description]

- This option generates an error when the SAVE or RSTR instruction dedicated to the register bank save function is written.

-create_unfilled_area [V2.03.00 or later]

< Assembler Command Options / Object Options >

[Format]

```
-create_unfilled_area
```

[Description]

- When a **Motorola S-record file** (**<name>.mot**) or **Hex file** (**<name>.hex**) is output, this option blocks spaces created by **.OFFSET** directives in the assembly language being filled with output data.
- When using this option, specify it when using the **ccrx** or **asrx** command to create an object file (**<name>.obj**) as well as when using the **rlink** command to create a **Motorola S-record file** or **Hex file**.

[Remarks]

- This option is available in V2.03 and later versions of this compiler.
- When this option is used, symbols in the format shown below*1 will be added for each .**OFFSET** directive.

```
__$_<FileName>_<SectionName>_<IDNumber>s__unfilled_area
__$_<FileName>_<SectionName>_<IDNumber>e__unfilled_area
```

Here, the name of the source file, section, and a number in a sequence starting from 1 are entered as **<FileName>**, **<SectionName>**, and **<IDNumber>**, respectively.

Note
*1) Since symbols in this format are reserved, they cannot be directly included in your source code.

List Options

< Assembler Command Options / List Options >

The following list options are available.

- -listfile
- -nolistfile
- -show

-listfile

< Assembler Command Options / List Options >

[Format]

```
-listfile[=<file name>]
```

- [Default]
  If this option is not specified, no assemble list file is output.

[Description]

- When the **listfile** option is specified, an assemble list file is output. The name of the file can also be specified.
- <file name> should be specified according to the rules described in the Naming Files section.
- If <file name> is not specified in the **listfile** option, the source file name with the extension replaced with **.lst** is used as the source list file name.

-nolistfile

< Assembler Command Options / List Options >

[Format]

```
-nolistfile
```

- [Default]
  If this option is not specified, no assemble list file is output.

[Description]

- When the **nolistfile** option is specified, no assemble list file is output.

-show

< Assembler Command Options / List Options >

[Format]

```
-show=<sub>[,...]
      <sub>: { conditionals | definitions | expansions }
```

[Description]

- This option specifies the contents of the list file to be output by the assembler. The following output types can be specified as <sub>.

Table 2.12 Output Types Specifiable for show Option

Output Type	Description
conditionals	The statements for which the specified condition is not satisfied in conditional assembly are also output to a source list file.
definitions	The information before replacement specified by **.DEFINE** is output to a source list file.
expansions	The macro expansion statements are output to a source list file.

Microcontroller Options

< Assembler Command Options / Microcontroller Options >

The following microcontroller options are available.

- -isa
- -cpu
- -endian
- -fint_register
- -base
- -patch
- -pic
- -pid
- -nouse_pid_register

-isa

< Assembler Command Options / Microcontroller Options >

[Format]

```
-isa={ rxv1 | rxv2 | rxv3 }
```

- [Default]
  The default for this option is determined based on the environment variable **ISA_RX**.

[Description]

- This option specifies an instruction-set architecture for use in generating object files.

[Remarks]

- When neither the **-nofpu** nor **-fpu** option has been selected, specifying this option automatically selects the **-fpu** option.
- Omitting this option when neither the **-cpu** option nor one of the environment variables (CPU_RX or ISA_RX) is specified will lead to an error.
- When the **-cpu** option and this option are specified simultaneously, an error will occur.

-cpu

< Assembler Command Options / Microcontroller Options >

[Format]

```
-cpu={ rx600 | rx200 }
```

- [Default]
  The default for this option is determined based on the environment variable **CPU_RX**.

[Description]

- This option specifies the CPU type for the instruction code to be generated.
- When **-cpu=rx600** is specified, a relocatable file for the RX600 Series is generated.
- When **-cpu=rx200** is specified, a relocatable file for the RX200 Series is generated.

[Remarks]

- This option is for compatibility with earlier products.
- For upcoming RX-family MCUs, the **isa** option will be used instead of the **cpu** option to select an instruction-set architecture. In developing new applications, use the **isa** option where possible.
- The **cpu** option can be replaced by the **-isa**, **-fpu** and **-nofpu** options as follows.
- -cpu=rx600 ==> -isa=rxv1 -fpu
- -cpu=rx200 ==> -isa=rxv1 -nofpu
- Omitting the **cpu** option will lead to an error if neither the **-isa** option nor one of the environment variables (CPU_RX or ISA_RX) is specified.
- The **-cpu** and **-isa** options cannot be specified at the same time.

-endian

< Assembler Command Options / Microcontroller Options >

[Format]

```
-endian={ big  |  little }
```

- [Default]
  The default for this option is **endian=little**.

[Description]

- When **endian=big** is specified, data bytes are arranged in big endian.
  When **endian=little** is specified, data bytes are arranged in little endian.

-fint_register

< Assembler Command Options / Microcontroller Options >

[Format]

```
-fint_register = {0 | 1 | 2 | 3 | 4 }
```

- [Default]
  The default for this option is **fint_register=0**.

[Description]

- This option outputs to the relocatable file the information about the general registers that are specified to be used only for fast interrupts through the same-name option in the compiler.

[Remarks]

- Be sure to set this option to the same value for all assembly processes in the project. If a different setting is made, correct operation is not guaranteed.
- Do not use a general register dedicated to fast interrupts for other purposes in assembly-language files. If such a register is used for any other purpose, correct operation is not guaranteed.
- If a register specified by this option is also specified by the **base** option, an error will be output.

-base

< Assembler Command Options / Microcontroller Options >

[Format]

```
-base = {   rom = <register>
          | ram = <register>
          | <address> = <register>}
                        <register> = {R8 to R13}
```

[Description]

- This option outputs to the relocatable file the information about the general register that is specified to be used only as a base address register through the same-name option in the compiler.

[Remarks]

- Be sure to set this option to the same value for all assembly processes in the project. If a different setting is made, correct operation is not guaranteed.
- Do not use a general register specified by this option for other purposes than a base address register. If such a register is used for any other purpose, correct operation is not guaranteed.
- If a single general register is specified for different areas, an error will be output.
- If a general register specified by the **fint_register** option is also specified by this option, an error will be output.

-patch

< Assembler Command Options / Microcontroller Options >

[Format]

```
-patch = { rx610 }
```

[Description]

- This option is used to avoid a problem specific to the CPU type.
- When **-patch=rx610** is specified, the **MVTIPL** instruction which causes a problem in the RX610 Group is handled as an undefined instruction. The **MVTIPL** instruction will not be recognized as an instruction and the error message E0552113 will be output.

-pic

< Assembler Command Options / Microcontroller Options >

[Format]

```
-pic
```

- [Default]
  This option generates a relocatable object indicating that code was generated with the PIC function disabled.

[Description]

- This option generates a relocatable object indicating that code was generated with the PIC function enabled.

[Remarks]

- Even if code conflicting with this option is written in the assembly code, it will not be checked.
- A relocatable object with the PIC function enabled cannot be linked with a relocatable object with the PIC function disabled.
- For the PIC function, also refer to Usage of PIC/PID Function.

-pid

< Assembler Command Options / Microcontroller Options >

[Format]

```
-pid[={ 16 | 32 }]
```

- [Default]
  This option generates a relocatable object indicating that code was generated with the PID function disabled.

[Description]

- This option generates a relocatable object indicating that code was generated with the PID function enabled.
- <PID register>
  - Based on the rules in the following table, one register from among R9 to R13 is selected according to the specification of the **fint_register** option. If the **fint_register** option is not specified, R13 is selected.

Table 2.13 Correspondences between fint_register Options and PID Registers

fint_register Option	PID Register
No **fint_register** specification	R13
fint_register = 0	
fint_register = 1	R12
fint_register = 2	R11
fint_register = 3	R10
fint_register = 4	R9

  - The PID register can be used only for the purpose of PID access.
- <Parameters>
  - The meaning of a parameter is the same as that for the compiler option with the same name.

[Remarks]

- Even if code conflicting with PID is written in the assembly code, it will not be checked.
- A relocatable object with the PID function enabled cannot be linked with a relocatable object with the PID function disabled.
- If a PID register specified by the **pid** option is also specified by the **base** option, error F0553111 will be output.
- If the **pid** option and **nouse_pid_register** option are selected simultaneously, error F0553103 will be output.
- For the PID function, also refer to Usage of PIC/PID Function.

-nouse_pid_register

< Assembler Command Options / Microcontroller Options >

[Format]

```
-nouse_pid_register
```

[Description]

- This option generates a relocatable object that was generated without using the PID register.
- If the PID register is used in the assembly-language source file, error message E0552058 will be output. Specifying this option, however, does not lead to an error if a substitute register defined in the assembler specifications is used as the PID register.
- A master program called by an application program in which the PID function is enabled needs to be assembled with this option. At this time, if the **fint_register** option is selected in the application program, the same parameter **fint_register** should also be selected in the master program.

[Remarks]

- If the **nouse_pid_register** option and **pid** option are selected simultaneously, error F0553103 will be output.
- If a register specified by the **nouse_pid_register** option is also specified by the **base** option, error F0553112 will be output.
- For the PID function, also refer to Usage of PIC/PID Function.

Other Options

< Assembler Command Options / Other Options >

The following other options are available.

- -logo
- -nologo
- -subcommand
- -euc
- -sjis
- -latin1
- -big5
- -gb2312
- -utf8 [V2.04.00 or later]

-logo

< Assembler Command Options / Other Options >

[Format]

```
-logo
```

- [Default]
  The copyright notice is output.

[Description]

- The copyright notice is output.

-nologo

< Assembler Command Options / Other Options >

[Format]

```
-nologo
```

- [Default]
  The copyright notice is output.

[Description]

- When the **nologo** option is specified, output of the copyright notice is disabled.

-subcommand

< Assembler Command Options / Other Options >

[Format]

```
-subcommand=<subcommand file name>
```

[Description]

- When the **subcommand** option is specified, the assembler options specified in a subcommand file are used at assembler startup. Specify options in a subcommand file in the same format as in the command line.

[Example]

- Contents of subcommand file **opt.sub**:

```
-listfile
-debug
```

- Command line specifications:
- When options are specified in the command line as shown (1) below, the assembler interprets them as shown in (2).

```
(1) asrx -endian=big -subcommand=opt.sub test.src
(2) asrx -endian=big -listfile -debug test.src
```

-euc

< Assembler Command Options / Other Options >

[Format]

```
-euc
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **EUC** code.

-sjis

< Assembler Command Options / Other Options >

[Format]

```
-sjis
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

-latin1

< Assembler Command Options / Other Options >

[Format]

```
-latin1
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **ISO-Latin1** code.

-big5

< Assembler Command Options / Other Options >

[Format]

```
-big5
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **Big5** code.

-gb2312

< Assembler Command Options / Other Options >

[Format]

```
-gb2312
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in **SJIS** code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in **GB2312** code.

-utf8 [V2.04.00 or later]

< Assembler Command Options / Other Options >

[Format]

```
-utf8
```

- [Default]
  This option specifies the character code to handle the characters in strings, character constants, and comments in SJIS code.

[Description]

- This option specifies the character code to handle the characters in strings, character constants, and comments in UTF-8 code.

2.5.3 Optimizing Linkage Editor (rlink) Options

Classification	Option	Description
Input Options	-Input	Specifies relocatable files.
	-library	Specifies library files.
	-binary	Specifies binary files.
	-define	Specifies symbol definitions.
	-entry	Specifies an entry symbol or entry address.
	-noprelink	Selects non-initiation of the prelinker.
	-allow_duplicate_module_name [V3.02.00 or later]	This option allows multiple same module names to be specified.

Classification	Option	Description
Output Options	-form	Selects the output file format.
	-debug	Debugging information is output to load module files.
	-sdebug	Debugging information is output to the **.dbg** file.
	-nodebug	Debugging information is not output.
	-record	Selects the record size.
	-end_record [V2.07.00 or later]	This option specifies the end record.
	-rom	Specifies the section mapping from ROM to RAM.
	-output	Specifies the names of files to be output.
	-map	Outputs an external symbol-allocation information file.
	-space	Data are output to fill unused ranges of memory.
	-message	Information-level messages are output.
	-nomessage	The output of messages is disabled.
	-msg_unused	Messages are output to indicate the presence of externally defined symbols to which there is no reference.
	-byte_count	Specifies the number of bytes in a data record.
	-fix_record_length_and_alignment [V2.08.00 or later]	Fixes the format of data records to be output.
	-crc	Specifies the format for output of the CRC code.
	-padding	Padding data are included at the end of each section.
	-vectn	Assigns an address to the specified vector number in the variable vector table (for the RX Family and M16C Family).
	-vect	Assigns an address to an unused area in the variable vector table (for the RX Family and M16C Family).
	-split_vect [V3.00.00 or later]	Generates split vector table sections.
	-jump_entries_for_pic	Outputs a jump table file (for the PIC function of the RX Family).
	-cfi [Professional Edition only] [V2.08.00 or later]	Generates the function list for use in detecting illegal indirect function calls.
	-cfi_add_func [V2.08.00 or later]	Specifies the symbol or address of a function to be added to the function list for use in detecting illegal indirect function calls.
	-cfi_ignore_module [V2.08.00 or later]	Specifies modules which are to be exempted from checking against the function list for use in detecting illegal indirect function calls.
	-create_unfilled_area [V2.03.00 or later]	[To be supported by V2.03 and later versions] Makes spaces created by .OFFSET unfilled.
List Options	-list	A linkage list file is output.
	-show	Selects the contents to be output in the linkage list file.

Classification	Option	Description
Optimize Options	-optimize	Selects the items to be optimized at linkage.
	-nooptimize	Selects no optimization at linkage.
	-samesize	Specifies the minimum size for unification of the same codes.
	-symbol_forbid	Specifies symbols for which unreferenced symbol deletion is disabled.
	-samecode_forbid	Specifies symbols for which same code unification is disabled.
	-section_forbid	Specifies a section where optimization is disabled.
	-absolute_forbid	Specifies an address range where optimization is disabled.
	-ALLOW_OPTIMIZE_ENTRY_BLOCK [V3.06.00 or later]	Performs optimization on the areas that are allocated before the execution start symbol.
Section Options	-start	Specifies a section start address.
	-fsymbol	Specifies the section where an external defined symbol will be placed in the output file.
	-aligned_section	Specifies the section alignment value as 16 bytes.
Verify Options	-cpu	Checks addresses for consistency.
	-contiguous_section	Specifies sections that will not be divided.
Other Options	-s9	Selects the output of an s9 record at the end of the file.
	-stack	Selects the output of a stack information file.
	-compress	Debugging information are compressed.
	-nocompress	Debugging information are not compressed.
	-memory	Selects the amount of memory to be used in linkage.
	-rename	Specifies symbol names and section names to be modified.
	-lib_rename [V3.01.00 or later]	Changes the name of a symbol or section that was input from a library.
	-delete	Specifies symbol names and module names to be deleted.
	-replace	Specifies library modules to be replaced.
	-extract	Specifies modules to be extracted from library files.
	-strip	Debugging information is deleted from absolute files and library files.
	-change_message	Specifies changes to the levels of messages (information, warning, and error).
	-hide	Name information on local symbols is deleted.
	-total_size	The total sizes of sections after linkage are sent to standard output.
	-verbose [V3.03.00 or later]	This option displays detailed information in the standard error output.
Subcommand File Option	-subcommand	Specifies a file from which to include command options.

Classification	Option	Description
Options Other Than Above	-logo	Selects the output of copyright information.
	-nologo	Selects the non-output of copyright information.
	-end	Selects the execution of option strings specified before **END**.
	-exit	Specifies the end of option specification.

Input Options

< Optimizing Linkage Editor (rlink) Options / Input Options >

The following input options are available.

- -Input
- -library
- -binary
- -define
- -entry
- -noprelink
- -allow_duplicate_module_name [V3.02.00 or later]

-Input

< Optimizing Linkage Editor (rlink) Options / Input Options >

[Format]

```
-Input = <suboption>[{, | Δ}...]
         <suboption>: <file name>[ (<module name>[,...] ) ]
```

[Description]

- Specifies an input file. Two or more files can be specified by separating them with a comma (,) or space.
- Wildcards (* or ?) can also be used for the specification. String literals specified with wildcards are expanded in alphabetical order. Expansion of numerical values precedes that of alphabetical letters. Uppercase letters are expanded before lowercase letters.
- Specifiable files are object files output from the compiler or the assembler, and relocatable or absolute files output from the optimizing linkage editor. A module in a library can be specified as an input file using the format of <library name>(<module name>). The module name is specified without an extension.
- If an extension is omitted from the input file specification, **obj** is assumed when a module name is not specified and **lib** is assumed when a module name is specified.

[Examples]

```
input=a.obj lib1(e)        ; Inputs a.obj and module e in lib1.lib.
input=c*.obj               ; Inputs all .obj files beginning with c.
```

[Remarks]

- When **form=object** or **extract** is specified, this option is unavailable.
- When an input file is specified on the command line, **input** should be omitted.

-library

< Optimizing Linkage Editor (rlink) Options / Input Options >

[Format]

```
-library = <file name>[,...]
```

[Description]

- Specifies an input library file. Two or more files can be specified by separating them with a comma (,).
- Wildcards (* or ?) can also be used for the specification. String literals specified with wildcards are expanded in the alphabetical order. Expansion of numerical values precedes that of alphabetical letters. Uppercase letters are expanded before lowercase letters.
- If an extension is omitted from the input file specification, **lib** is assumed.
- If **form=library** or **extract** is specified, the library file is input as the target library to be edited.
- Otherwise, after the linkage processing between files specified for the input files are executed, undefined symbols are searched in the library file.
- The symbol search in the library file is executed in the following order: user library files with the library option specification (in the specified order), the system library files with the library option specification (in the specified order), and then the default library (environment variable **HLNK_LIBRARY1,2,3**).

[Examples]

```
library=a.lib,b    ; Inputs a.lib and b.lib.
library=c*.lib     ; Inputs all files beginning with c with the extension .lib.
```

-binary

< Optimizing Linkage Editor (rlink) Options / Input Options >

[Format]

```
-binary = <suboption>[,...]
          <suboption>: <file name>(<section name>
                          [:<boundary alignment>][/<section attribute>][,<symbol name>])
          <section attribute>: CODE | DATA
          <boundary alignment>: 1 | 2 | 4 | 8 | 16 | 32 (default: 1)
```

[Description]

- Specifies an input binary file. Two or more files can be specified by separating them with a comma (,).
- If an extension is omitted for the file name specification, **bin** is assumed.
- Input binary data is allocated as the specified section data. The section address is specified with the **start** option. That section cannot be omitted.
- When a symbol is specified, the file can be linked as a defined symbol. For a variable name referenced by a C/C++ program, add an underscore (_) at the head of the reference name in the program.
- The section specified with this option can have its section attribute and boundary alignment specified.
- CODE or DATA can be specified for the section attribute.
- When section attribute specification is omitted, the write, read, and execute attributes are all enabled by default.
- A boundary alignment value can be specified for the section specified by this option. A power of 2 can be specified for the boundary alignment; no other values should be specified.
- When the boundary alignment specification is omitted, 1 is used as the default.

[Examples]

```
input=a.obj
start=P,D*/200
binary=b.bin(D1bin),c.bin(D2bin:4,_datab)
form=absolute
```

- Allocates **b.bin** from 0x200 as the **D1bin** section.
- Allocates **c.bin** after **D1bin** as the **D2bin** section (with boundary alignment = 4).
- Links **c.bin** data as the defined symbol _**datab**.

[Remarks]

- When **form={object | library}** or **strip** is specified, this option is unavailable.
- If no input object file is specified, this option cannot be specified.

-define

< Optimizing Linkage Editor (rlink) Options / Input Options >

[Format]

```
-define = <suboption>[,...]
          <suboption>: <symbol name>={<symbol name> | <numerical value>}
```

[Description]

- Defines an undefined symbol forcedly as an externally defined symbol or a numerical value.
- The numerical value is specified in the hexadecimal notation. If the specified value starts with a letter from A to F, symbols are searched first, and if no corresponding symbol is found, the value is interpreted as a numerical value. Values starting with 0 are always interpreted as numerical values.
- If the specified symbol name is a C/C++ variable name, add an underscore (_) at the head of the definition name in the program. If the symbol name is a C++ function name (except for the **main** function), enclose the definition name with the double-quotes including parameter strings. If the parameter is **void**, specify as "<function name>( )".

[Examples]

```
define=_sym1=data  ; Defines _sym1 as the same value as the externally defined symbol
data.
define=_sym2=4000  ; Defines _sym2 as 0x4000.
```

[Remarks]

- When **form={object | relocate | library}** is specified, this option is unavailable.

-entry

< Optimizing Linkage Editor (rlink) Options / Input Options >

[Format]

```
-entry = {<symbol name> | <address>}
```

[Description]

- Specifies the execution start address with an externally defined symbol or address.
- The address is specified in hexadecimal notation. If the specified value starts with a letter from A to F, symbols are searched first, and if no corresponding symbol is found, the value is interpreted as an address. Values starting with 0 are always interpreted as addresses.
- For a C function name, add an underscore (_) at the head of the definition name in the program. For a C++ function name (except for the **main** function), enclose the definition name with double-quotes in the program including parameter strings. If the parameter is **void**, specify as "<function name>()".
- If the **entry** symbol is specified at compilation or assembly, this option precedes the entry symbol.

[Examples]

```
entry=_main        ; Specifies main function in C/C++ as the execution start address.
entry="init()"     ; Specifies init function in C++ as the execution start address.
entry=100          ; Specifies 0x100 as the execution start address.
```

[Remarks]

- When **form={object | relocate | library}** or **strip** is specified, this option is unavailable.
- When optimization with undefined symbol deletion (**optimize=symbol_delete**) is specified, the execution start address should be specified. If it is not specified, the specification of the optimization with undefined symbol deletion is unavailable. Optimization with undefined symbol deletion is not available when an address is specified with this option.
- If the address specified by the **-entry** option belongs to the list of sections for which the location address is specified by the **-start** option, optimization is disabled for the area from the location address specified by the **-start** option to the address specified by the **-entry** option.

-noprelink

< Optimizing Linkage Editor (rlink) Options / Input Options >

[Format]

```
-noprelink
```

- [Default]
  If this option is not specified, the prelinker is initiated.

[Description]

- Disables the prelinker initiation.
- The prelinker supports the functions to generate the C++ template instance automatically and to check types at run time. When the C++ template function and the run-time type test function are not used, specify the **noprelink** option to reduce the link time.

[Remarks]

- When **extract** or **strip** is specified, this option is unavailable.
- If **form=lib** or **form=rel** is specified while the C++ template function and run-time type test are used, do not specify **noprelink**.

-allow_duplicate_module_name [V3.02.00 or later]

< Optimizing Linkage Editor (rlink) Options / Input Options >

[Format]

```
-allow_duplicate_module_name
```

[Description]

- This option allows multiple input files with the same module name to be specified to generate a library.
- If the library already contains a module having the same name with other modules to be registered in the library, the other modules are renamed by adding a postfix number ".<N>".
- <N> is assigned a number as a unique module name in the generating library. If can't assigned a unique number, The linker will output the error message and quit.

[Examples]

- To generate a library a.lib from multiple input files having the same module name (mod), describe as:

```
> rlink -allow_duplicate_module_name -form=lib -output=a.lib b\mod.obj c\mod.obj
d\mod.obj
```

The command line above leads to generate a library a.lib containing the following modules:

- mod (originally b\mod.obj)
- mod.1 (originally c\mod.obj)
- mod.2 (originally d\mod.obj)

[Remarks]

- If the **-form={ object|absolute|relocate|hexadecimal|stype|binary }**, **-strip**, or **-extract** option is specified, this option will be invalid.

Output Options

< Optimizing Linkage Editor (rlink) Options / Output Options >

The following output options are available.

- -form
- -debug
- -sdebug
- -nodebug
- -record
- -end_record [V2.07.00 or later]
- -rom
- -output
- -map
- -space
- -message
- -nomessage
- -msg_unused
- -byte_count
- -fix_record_length_and_alignment [V2.08.00 or later]
- -crc
- -padding
- -vectn
- -vect
- -split_vect [V3.00.00 or later]
- -jump_entries_for_pic
- -cfi [Professional Edition only] [V2.08.00 or later]
- -cfi_add_func [V2.08.00 or later]
- -cfi_ignore_module [V2.08.00 or later]
- -create_unfilled_area [V2.03.00 or later]

-form

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-form = {Absolute | Relocate | Object | Library[={S | U}]} | Hexadecimal | Stype |
Binary}
```

- [Default]
  When this option is omitted, the default is **form=absolute**.

[Description]

- Specifies the output format.
- Table 2.14 lists the suboptions.

Table 2.14 Suboptions of form Option

No	Suboption	Description
1	absolute	Outputs an absolute file
2	relocate	Outputs a relocatable file
3	object	Outputs an object file. This is specified when a module is extracted as an object file from a library with the **extract** option.
4	library	Outputs a library file. When **library=s** is specified, a system library is output. When **library=u** is specified, a user library is output. Default is **library=u**.
5	hexadecimal	Outputs a **HEX** file. For details of the HEX format, refer to HEX File Format.
6	stype	Outputs an **S**-type file. For details of the **S**-type format, refer to S-Type File Format.
7	binary	Outputs a binary file.

[Remarks]

Table 2.15 shows relations between output formats and input files or other options.

Table 2.15 Relations Between Output Format and Input File or Other Options

No	Output Format	Specified Option	Enabled File Format	Specifiable Option[Note1]
1	Absolute	**strip** specified	Absolute file	input, output
		Other than above	Object file Relocatable file Binary file Library file	input, library, binary, debug/nodebug, sdebug, cpu, start, rom, entry, output, map, hide, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, section_forbid, absolute_forbid, compress, rename, lib_rename, delete, define, fsymbol, stack, noprelink, memory, msg_unused, show={symbol,reference,xreference,total_size, vector,relocation_attribute} jump_entries_for_pic, aligned_section, padding, vectn, vect, split_vect
2	Relocate	**extract** specified	Library file	library, output
		Other than above	Object file Relocatable file Binary file Library file	input, library, debug/nodebug, output, hide, rename, lib_rename, delete, noprelink, msg_unused, show=symbol,reference,xreference,total_size
3	Object	**extract** specified	Library file	library, output
4	Hexadecimal Stype Binary		Object file Relocatable file Binary file Library file	input, library, binary, cpu, start, rom, entry, output, map, space, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, section_forbid, absolute_forbid, rename, lib_rename, delete, define, fsymbol, stack, noprelink, record, end_record, s9[Note 2], byte_count[Note3], fix_record_length_and_align, memory, msg_unused, show=symbol,reference,xreference,total_size, vector, jump_entries_for_pic, aligned_section, padding, vectn, vect, split_vect
			Absolute file	input, output, record, end_record, s9[Note2], byte_count[Note3], fix_record_length_and_align, show=symbol, reference, xreference
5	Library	**strip** specified	Library file	library, output, memory[Note4], show=symbol, section
		extract specified	Library file	library, output
		Other than above	Object file Relocatable file	input, library, output, hide, rename, delete, replace, noprelink, memory[Note4], show=symbol, section, allow_duplicate_module_name

Notes 1. **message/nomessage**, **change_message**, **logo/nologo**, **form**, **list**, and **subcommand** can always be specified.

Notes 2. **s9** can be used only when **form=stype** is specified for the output format.

Notes 3. **byte_count** can be used only when **form=hexadecimal** or **form=stype** is specified for the output format.

Notes 4. **memory** cannot be used when **hide** is specified.

-debug

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-debug
```

- [Default]
  When this option is omitted, debugging information is output to the output file.

[Description]

- When **debug** is specified, debugging information is output to the output file.
- If **debug** is specified and if two or more files are specified to be output with **output**, they are interpreted as **sdebug** and debugging information is output to **<first output file name>.dbg**.

[Remarks]

- When **form={object | library | hexadecimal | stype | binary}**, **strip** or **extract** is specified, this option is unavailable.

-sdebug

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-sdebug
```

- [Default]
  When this option is omitted, debugging information is output to the output file.

[Description]

- When **sdebug** is specified, debugging information is output to **<output file name>.dbg** file.
- If **sdebug** and **form=relocate** are specified, **sdebug** is interpreted as **debug**.

[Remarks]

- When **form={object | library | hexadecimal | stype | binary}**, **strip** or **extract** is specified, this option is unavailable.

-nodebug

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-nodebug
```

- [Default]
  When this option is omitted, debugging information is output to the output file.

[Description]

- When **nodebug** is specified, debugging information is not output.

[Remarks]

- When **form={object | library | hexadecimal | stype | binary}**, **strip** or **extract** is specified, this option is unavailable.

-record

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-record = { H16 | H20 | H32 | S1 | S2 | S3 }
```

- [Default]
  When this option is omitted, various data records are output according to each address.

[Description]

- Outputs data with the specified data record regardless of the address range.
- If there is an address that is larger than the specified data record, the appropriate data record is selected for the address.

[Remarks]

- This option is available only when **form=hexadecimal** or **stype** is specified.

-end_record [V2.07.00 or later]

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-end_record = { S7 | S8 | S9 }
```

[Description]

- This option specifies the type of the end record of a Motorola S-record file.
- When the entry point address is larger than the specified address field, select the type of end record that suits the address of the entry point.
- When this option is omitted, the end record that is output will suit the address of the entry point.

[Example]

```
rlink a.obj b.obj -end_record=S7 -form=stype -output=c.mot
```

- An S-type end record that is interpreted as a 32-bit address (S7) is output regardless of the address range.

[Remarks]

- When -form={stype} is not also specified, an error message is output in response to this option and execution is terminated.

-rom

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-rom = <suboption>[,...]
       <suboption>:  <ROM section name>=<RAM section name>
```

[Description]

- Reserves ROM and RAM areas in the initialized data area and relocates a defined symbol in the ROM section with the specified address in the RAM section.
- Specifies a relocatable section including the initial value for the ROM section.
- Specifies a nonexistent section or relocatable section whose size is 0 for the RAM section.
- A wildcard symbol (*) can be used in ROM and RAM section names. [V3.06.00 or later]
    - If the name of a relocatable ROM section with the initial value matches the wildcard expression of the ROM section name, the name is processed as a RAM section name. At this time, a wildcard symbol (*) in the RAM section name is replaced with the part that matches the wildcard symbol (*) in the ROM section name.

      Example When there are three ROM sections (D, D_1, and D_2) and **-rom=D*=R*** is specified, three RAM sections (R, R_1, and R_2) are generated.

      Note The RAM section name after replacement must be handled appropriately by using, for example, the **-start** option.
    - Multiple wildcard symbols (*) can be specified. The number of wildcard symbols must match between *ROMsection* and *RAMsection*.

      Example

      ```
      -rom=D*=R*        # No problem
      -rom=D*=*R*       # Error due to too many wildcard symbols in RAMsection
      ```
    - If a section having the same name as the one generated by replacement already exists, an error occurs.

[Examples]

```
rom=D=R
start=D/100,R/8000
```

- Reserves **R** section with the same size as **D** section and relocates defined symbols in **D** section with the **R** section addresses.

[Remarks]

- When **form={object | relocate | library}**or **strip** is specified, this option is unavailable.

-output

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-output = <suboption>[,...]
          <suboption>:  {<file name> | <file name>=<output range> |
                         <file name>=<output range>/<load address> |
                         <file name>=/<load address>}
          <output range>:  {<start address>-<end address> | <section name>[:...]}
```

- [Default]
  When this option is omitted, the default is <first input file name>.<default extension>.
  The default extensions are as follows:
  form=absolute: abs, form=relocate: rel, form=object: obj, form=library: lib, form=hexadecimal: hex, form=stype: mot, form=binary: bin

[Description]

- Specifies an output file name. When **form=absolute**, **hexadecimal**, **stype**, or **binary** is specified, two or more files can be specified. An address is specified in the hexadecimal notation. If the specified data starts with a letter from A to F, sections are searched first, and if no corresponding section is found, the data is interpreted as an address. Data starting with 0 are always interpreted as addresses.
- [V3.00.00 or later] If a load address is specified, when outputting an Intel HEX file or Motorola S-record file, the first load address in the file is changed to the specified value.

[Examples]

```
output=file1.abs=0-ffff,file2.abs=10000-1ffff
```

- Outputs the range from 0 to 0xffff to **file1.abs** and the range from 0x10000 to 0x1ffff to **file2.abs**.

  ```
  output=file1.abs=sec1:sec2,file2.abs=sec3
  ```

- Outputs the **sec1** and **sec2** sections to **file1.abs** and the **sec3** section to **file2.abs**.

[Remarks]

- When a file is output in section units while the CPU type is RX Family in big endian, the section size should be a multiple of 4.
- [V3.00.00 or later] A load address can be specified only when **form=hexadecimal** or **form=stype** is specified.

-map

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-map [= <file name>]
```

[Description]

- Outputs the external-symbol-allocation information file that is used by the compiler in optimizing access to external variables.
- When <file name> is not specified, the file has the name specified by the **output** option or the name of the first input file, and the extension **bls**.
- If the order of the declaration of variables in the external-symbol-allocation information file is not the same as the order of the declaration of variables found when the object was read after compilations, an error will be output.
- In the following case, the linker outputs the external variable allocation information file and, when the **-list** option is specified, outputs the list file. After that, the linker terminates operation normally. Note that the linker does not output a load module file in this case.
    - When a program section allocation address exceeds the allowable address range:

  In the external variable allocation information file, information regarding only the symbols and sections allocated within the allowable areas are output. [V2.06 or later]

[Remarks]

- This option is valid only when **form={absolute | hexadecimal | stype | binary}** is specified.

-space

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-space [= {<numerical value> | Random}]
```

[Description]

- This option fills the vacant area of the output range with user-specified data.
- Fills the unused areas in the output ranges with random values or a user-specified hexadecimal value.
- The way of filling unused areas differs with the output range specification as follows.
    - When the **output** option is used to specify sections as the range for output:
        - The specified numerical value is output to unused areas between the specified sections.
    - When the **output** option is used to specify a range of addresses as the range for output:
        - The specified numerical value is output to unused areas within the specified address range.
    - When the **fix_record_length_and_align** option is specified:
        - The specified numerical value is output to an unused area at the top of a section, which starts at an address that can be divided by the alignment number.
        - The specified numerical value is output when the end of a section does not reach the specified record length.

[Remarks]

- When no suboption is specified by this option, unused areas are not filled with values.
- This option is only available when the **form={ binary | stype | hexadecimal }** or **fix_record_length_and_align** option is specified.

-message

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-message
```

[Description]

- When **message** is specified, information-level messages are output.
- When this option is omitted, the output of information-level messages is disabled.

-nomessage

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-nomessage [=<suboption>[,...]]
            <suboption>:  <error number>[-<error number>]
```

- [Default]
  When this option is omitted, the output of information-level messages is disabled.

[Description]

- When **nomessage** is specified, the output of information-level messages is disabled. If an error number is specified, the output of the error message with the specified error number is disabled. A range of error message numbers to be disabled can be specified using a hyphen (-).
- Each error number consists of a component number (05), phase (6), and a four-digit value (e.g. 0004 in the case of M0560004). If the four-digit section has leading zeroes, e.g. before the 4 in the case of M0560004, these can be omitted.
- If a warning or error level message number is specified, the message output is disabled assuming that **change_message** has changed the specified message to the information level.

[Examples]

- Messages of L0004, L0200 to L0203, and L1300 are disabled to be output.

```
nomessage=4,200-203,1300
```

-msg_unused

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-msg_unused
```

[Description]

- Notifies the user of the externally defined symbol which is not referenced during linkage through an output message.

[Examples]

```
rlink -msg_unused a.obj
```

[Remarks]

- When an absolute file is input, this option is invalid.
- To output a message, the **message** option must also be specified.
- The linkage editor may output a message for the function that was inline-expanded at compilation. To avoid this, add a **static** declaration for the function definition.
- In any of the following cases, references are not correctly analyzed so that information shown by output messages will be incorrect.
    - There are references to constant symbols within the same file.
    - There are branches to immediate subordinate functions when optimization is specified at compilation.

-byte_count

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-byte_count=<numerical value>
```

- [Default]
  The default values are FF (hexadecimal) for Intel HEX-format files and 10 (hexadecimal) for Motorola S-format files.

[Description]

- This option is used to specify the length of data records in Intel HEX-format files or Motorola S-format files to be generated.
- Values from 01 to FF (hexadecimal) are specifiable for Intel HEX-format files.
- The following ranges of values are specifiable for Motorola S-format files.
  - S1 records: 01 to FC (hexadecimal)
  - S2 records: 01 to FB (hexadecimal)
  - S3 records: 01 to FA (hexadecimal)

[Examples]

Specifying 16 bytes (10 in hexadecimal) as the length of data records

```
-byte_count=10
```

[Remarks]

- This option is invalid for file formats other than Intel HEX type (**form=hex**) or Motorola S-record type (**form=stype**).

-fix_record_length_and_alignment [V2.08.00 or later]

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-fix_record_length_and_align=<alignment number>
```

[Description]

- This option is used to output an Intel HEX- or Motorola S-format file with records of a fixed length starting from the address that has alignment with the specified number.
- The address of the first record to be output should be less than or equal to the first address of a section and be the largest number that can be divided by the specified alignment number.
- The specified numerical value or default value for the parameter of the **byte_count** option will be used as the length of the records.
- Since the length of records is fixed, each record may include data for more than one section.
- In unused areas, the value specified by the **space** option will be output. If the **space** option is not specified, 0 (with the **crc** option not specified) or FF (with the **crc** option specified) as the default value will be output.

[Examples]

Starting the output of records from an address that can be divided by 8, with the length of each record fixed to 16 bytes (10 in hexadecimal)

```
rlink a.obj b.obj -form=hexadecimal -byte_count=10 -fix_record_length_and_align=8
```

[Remarks]

- This option is invalid for file formats other than Intel HEX type (**form=hex**) or Motorola S-record type (**form=stype**).

-crc

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-CRc = <suboption>
       <suboption>: <address where the result is output>=<target range>
                         [/<Operation Method>][<initial value>][:<endian>]
                    <address where the result is output>: <address>
                    <target range>: { <start address>-<end address> |
                                      <section> }[,...]
                    <Operation Method>: { CCITT | 16-CCITT-MSB |
                                               16-CCITT-MSB-LITTLE-4 |
                                               16-CCITT-MSB-LITTLE-2 | 16-CCITT-LSB |
                                               16 | SENT-MSB | 32-ETHERNET }
                    <initial value>: <initial value>
                    <endian>: {BIG | LITTLE}[-<size>-<offset>]
```

[Description]

- CRC (cyclic redundancy check) operation is done for the specified range of section data in the order from the lower to the higher addresses, and the operation result is output to the specified output address in the specified endian mode.
- Specify one of the following as the operation method. If the specification of the operation method is omitted, operation is performed assuming that CCITT has been specified.

Table 2.16 List of Operation Methods

Operation Method	Description
CCITT	The result of CRC-16-CCITT operation is obtained with the MSB first, an initial value of 0xFFFF, and inverse of XOR performed. The generator polynomial is $x^{16}+x^{12}+x^5+1$.
16-CCITT-MSB [V2.04.00 or later]	The result of CRC-16-CCITT operation is obtained with the MSB first. The generator polynomial is $x^{16}+x^{12}+x^5+1$.
16-CCITT-MSB-LITTLE-4 [V2.04.00 or later]	The input is handled in little endian in 4-byte units and the result of CRC-16-CCITT operation is obtained with the MSB first. The generator polynomial is $x^{16}+x^{12}+x^5+1$.
16-CCITT-MSB-LITTLE-2 [V2.04.00 or later]	The input is handled in little endian in 2-byte units and the result of CRC-16-CCITT operation is obtained with the MSB first. The generator polynomial is $x^{16}+x^{12}+x^5+1$.
16-CCITT-LSB [V2.04.00 or later]	The result of CRC-16-CCITT operation is obtained with the LSB first. The generator polynomial is $x^{16}+x^{12}+x^5+1$.
16	The result of CRC-16 operation is obtained with the LSB first. The generator polynomial is $x^{16}+x^{15}+x^2+1$.
SENT-MSB [V2.04.00 or later]	The input is handled in little endian in the lower 4-bit units of one byte and the result of SENT-compliant CRC operation is obtained with the MSB first and an initial value of 0x5. The generator polynomial is $x^4+x^3+x^2+1$.
32-ETHERNET [V2.04.00 or later]	The result of CRC-32-ETHERNET operation is obtained with an initial value of 0xFFFFFFFF, inverse of XOR performed, and the bits reversed. The generator polynomial is $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$.

- The range of values that can be specified for <initial value> is from 0x0 to 0xFFFFFFFF when the operation method is 32-ETHERNET, from 0x0 to 0xF when the operation method is SENT-MSB, and from 0x0 to 0xFFFF for the other cases.
- When <initial value> is omitted, operation is performed on the assumption that 0x5 has been specified for the operation method of SENT-MSB, 0xFFFF for CCITT, 0xFFFFFFFF for 32-ETHERNET, and 0x0 for other cases.
- The operation result is output to the specified output address by writing at the offset location from the beginning of the area allocated by size in the byte order specified with BIG or LITTLE. 0 is output from the beginning of the allocated area until immediately before the offset location.
- When the size and offset are omitted, the size is assumed to be 2 bytes and the offset is assumed to be 0.
- When the space option is not specified, space=FF is assumed for CRC operation for the unused areas in the operation range. Note that 0xFF is only assumed for CRC operation for the unused areas, but the areas are not actually filled with 0xFF.
- Operation is done from the lower to the higher addresses of the specified operation range.
- If this option is specified more than once, the results of all the specified CRC operations will be output. [V3.05.00 or later]

[Example]

- rlink *.obj -form=stype -start=P1,P2/1000,P3/2000
  -crc=2FFE=1000-2FFD -output=out.mot=1000-2FFF

- **crc** option: -crc=2FFE=1000-2FFD
  - In this example, CRC will be calculated for the range from 0x1000 to 0x2FFD and the result will be output to address 0x2FFE.
  - When the **space** option has not been specified, **space=0xFF** is assumed for calculation of free areas within the target range.
- **output** option: -output=out.mot=1000-2FFF
  - Since the **space** option has not been specified, the free areas are not output to the **out.mot** file. 0xFF is used in CRC for calculation of the free areas, but will not be filled into these areas.

  Notes 1. The address where the result of CRC will be output cannot be included in the target range.

  Notes 2. The address where the result of CRC will be output must be included in the output range specified with the **output** option.

- rlink *.obj -form=stype -start=P1/1000,P2/1800,P3/2000
  -space=7F -crc=2FFE=1000-17FF,2000-27FF
  -output=out.mot=1000-2FFF

- **crc** option: -crc=2FFE=1000-2FFD,2000-27FF
  - In this example, CRC will be calculated for the two ranges, 0x1000 to 0x17FF and 0x2000 to 0x27FF, and the result will be output to address 0x2FFE.
  - Two or more non-contiguous address ranges can be selected as the target range for CRC.
- **space** option: -space=7F
  - The value of the **space** option (0x7F) is used for CRC in free areas within the target range.
- **output** option: -output=out.mot=1000-2FFF
  - Since the **space** option has been specified, the free areas are output to the **out.mot** file. 0x7F will be filled into the free areas.

  Notes 1. The order that CRC is calculated for the specified address ranges is not the order that the ranges have been specified. CRC proceeds from the lowest to the highest address.

  Notes 2. Even if you wish to use the **crc** and **space** options at the same time, the **space** option cannot be set as **random** or a value of 2 bytes or more. Only 1-byte values are valid.

- rlink *.obj -form=stype -start=P1,P2/1000,P3/2000
  -crc=1FFE=1000-1FFD,2000-2FFF
  -output=flmem.mot=1000-1FFF

- **crc** option: -crc=1FFE=1000-1FFD,2000-2FFF
  - In this example, CRC will be calculated for the two ranges, 0x1000 to 0x1FFD and 0x2000 to 0x2FFF, and the result will be output to address 0x1FFE.
  - When the **space** option has not been specified, **space=0xFF** is assumed for calculation of free areas within the target range.
- **output** option: -output=flmem.mot=1000-1FFF
  - Since the **space** option has not been specified, the free areas are not output to the **flmem.mot** file.
  - 0xFF is used in CRC for calculation of the free areas, but will not be filled into these areas.

- rlink *.obj -form=stype -start=.SEC1,.SEC2/1000,.SEC3/2000 -output=out.mot=1000-2FFF
  -crc=2FFC=1000-1FFF -crc=2FFE=2000-2FFB

- **crc** option (1): -crc=2FFC=1000-1FFF
  - In this example, CRC will be calculated for the range from 0x1000 to 0x1FFF, and the result will be output to address 0x2FFC.
- **crc** option (2): -crc=2FFE=2000-2FFB
  - In this example, CRC will be calculated for the range from 0x2000 to 0x2FFB, and the result will be output to address 0x2FFE.

[Remarks]

- When multiple load module files are input, the compiler outputs a warning message and ignores this option.
- This option is valid when the output format is **form={hexadecimal | stype | bin}**. For any other cases, an error is output and execution is terminated.
- When the space option is not specified and the operation range includes an empty area that is not output, 0xFF is assumed to be stored in the unused area during CRC operation.
- An error is output and execution is terminated if the CRC operation range includes an overlaid area.
- The following can be specified for the size and offset when specifying the endian. For any other cases, an error is output and execution is terminated.
  - LITTLE
  - LITTLE-2-0
  - LITTLE-4-0
  - BIG
  - BIG-2-0
  - BIG-4-0
- Sample Code: The sample code shown below is provided to check the result of CRC figured out by the **crc** option. The sample code program should match the result of CRC by rlink.
- When the selected operation method is **CRC-CCITT**:

```
typedef unsigned char    uint8_t;
typedef unsigned short   uint16_t;
typedef unsigned ong    uint32_t;

uint16_t CRC_CCITT(uint8_t *pData, uint32_t iSize)
{
    uint32_t    ui32_i;
    uint8_t     *pui8_Data;
    uint16_t    ui16_CRC = 0xFFFFu;

    pui8_Data = (uint8_t *)pData;

    for(ui32_i = 0; ui32_i < iSize; ui32_i++)
    {
        ui16_CRC  = (uint16_t)((ui16_CRC >> 8u) |
                              ((uint16_t)((uint32_t)ui16_CRC << 8u)));
        ui16_CRC ^= pui8_Data[ui32_i];
        ui16_CRC ^= (uint16_t)((ui16_CRC & 0xFFu) >> 4u);
        ui16_CRC ^= (uint16_t)((ui16_CRC << 8u) << 4u);
        ui16_CRC ^= (uint16_t)(((ui16_CRC & 0xFFu) << 4u) << 1u);
    }
    ui16_CRC = (uint16_t)( 0x0000FFFFul &
                              ((uint32_t)~(uint32_t)ui16_CRC) );
    return ui16_CRC;
}
```

- When the selected operation method is **CRC-16**:

```
#define POLYNOMIAL 0xa001 // Generated polynomial expression CRC-16

typedef unsigned char    uint8_t;
typedef unsigned short   uint16_t;
typedef unsigned long    uint32_t;

uint16_t CRC16(uint8_t *pData, uint32_t iSize)
{
    uint16_t crcdData = (uint16_t)0;
    uint32_t data = 0;
    uint32_t i,cycLoop;

    for(i=0;i<iSize;i++){
        data = (uint32_t)pData[i];
        crcdData = crcdData ^ data;
        for (cycLoop = 0; cycLoop < 8; cycLoop++) {
            if (crcdData & 1) {
                crcdData = (crcdData >> 1) ^ POLYNOMIAL;
            } else {
                crcdData = crcdData >> 1;
            }
        }
    }
    return crcdData;
}
```

-padding

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-padding
```

[Description]

- Fills in padding data at the end of a section so that the section size is a multiple of the boundary alignment of the section.
- The file name is <output file>.jmp.
- This option fills in padding data only in sections of an instruction, the const variable, and a variable with the initial value. This option does not apply to sections of variables that have no initial values.

[Examples]

```
-start=P,C/0 -padding
```

- When the boundary alignment of section **P** is 4 bytes, the size of section **P** is 0x06 bytes, the boundary alignment of section **C** is 1 byte, and the size of section **C** is 0x03 bytes, two bytes of padding data is filled in section **P** to make its size become 0x08 bytes and then linkage is performed.

```
-start=P/0,C/7 -padding
```

- When the boundary alignment of section **P** is 4 bytes, the size of section **P** is 0x06 bytes, the boundary alignment of section **C** is 1 byte, and the size of section **C** is 0x03 bytes, if two bytes of padding data is filled in section **P** to make its size become 0x08 bytes and then linkage is performed, error L2321 will be output because section **P** overlaps with section **C**.

[Remarks]

- The value of the created padding data is 0x00.
- Since padding is not performed to an absolute address section, the size of an absolute address section should be adjusted by the user.

-vectn

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-vectn = <suboption>[,...]
         <suboption>:  <vector number> = {<symbol> | <address>}
```

[Description]

- Assigns the specified address to the specified vector number in the variable vector table section.
- When this option is specified, a variable vector table section is created and the specified address is set in the table even if there is no interrupt function in the source code.
- Specify a decimal value from 0 to 255 for <vector number>.
- Specify the external name of the target function for <symbol>.
- Specify the desired hexadecimal address for <address>.
- [V3.00.00 or later] When **split_vect** is not specified, set a value in an unused area which is not specified with **vectn** according to the following priority.
  1. Value specified with the **vect** option
  2. If there is a defined symbol with the name (internal name) of "__dummy_int" in the link target, the address of that symbol
  3. If there is a defined symbol with the name (internal name) of "dummy_int" in the link target, the address of that symbol
  4. 0 for cases other than any of the above

  When **split_vect** is specified, a section for each vector number is not generated for an unused area which is not specified with **vectn**.

[Examples]

```
-vectn=30=_f1,31=0000F100 ;Specifies the _f1 address for vector
                          ;number 30 and 0x0f100 for vector number 31
```

[Remarks]

- This option is ignored when the user creates a variable vector table section in the source program because the variable vector table is not automatically created in this case.

-vect

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-vect={<symbol>|<address>}
```

[Description]

- Assigns the specified address to the vector number to which no address has been assigned in the variable vector table section.
- When this option is specified, a variable vector table section is created by the linkage editor and the specified address is set in the table even if there is no interrupt function in the source code.
- Specify the external name of the target function for <symbol>.
- Specify the desired hexadecimal address for <address>.
- The file name is <output file>.jmp.

[Remarks]

- This option is ignored when the user creates a variable vector table section in the source program because the variable vector table is not automatically created in this case.
- When the **{<symbol>|<address>}** specification is started with 0, the whole specification is assumed as an address.

-split_vect [V3.00.00 or later]

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-split_vect
```

[Description]

- This option generates vector table sections split by vector table number.
- A vector table section is not generated for an unused area of the vector table number.

[Example]

- To generate a vector table section "C$VECT14" for vector table number 14, code as:

```
-vectn=14=__dummy -split_vect
```

[Remarks]

- This option is invalid when the **-vect** option, **-form={object | relocate | library}** option, **-strip** option, or **-extract** option is specified at the same time.

-jump_entries_for_pic

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-jump_entries_for_pic=<section name>[,...]
```

[Description]

- Outputs an assembly-language source for a jump table to branch to external definition symbols in the specified section.
- The file name is <output file>.jmp.

[Examples]

- A jump table for branching to external definition symbols in the sections **sct2** and **sct3** is output to **test.jmp**.

```
jump_entries_for_pic=sct2,sct3
output=test.abs
```

- [Example of a file output to **test.jmp**]

```
;OPTIMIZING LINKAGE EDITOR GENERATED FILE 2009.07.19
    .glb _func01
    .glb _func02
    .SECTION P,CODE
_func01:
    MOV.L #1000H,R14
    JMP   R14
_func02:
    MOV.L #2000H,R14
    JMP   R14
    .END
```

[Remarks]

- This option is invalid when **form={object | relocate| library}** or **strip** is specified.
- The generated jump table is output to the **P** section.
- Only the program section can be specified for the type of section in the section name.

-cfi [Professional Edition only] [V2.08.00 or later]

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-cfi
```

- Interpretation when omitted
  The function list for use in detecting illegal indirect function calls is not generated.

[Description]

- This option selects generation of the function list for use in detecting illegal indirect function calls.
  For details on detecting illegal indirect function calls, refer to the item on the '-control_flow_integrity [Professional Edition only] [V2.08.00 or later]' compile option.
- Since the linker generates the function list for the C section, the C section must be specified with the **-start** option at the time of linking.
- When an object file is created with **-control_flow_integrity** specified at the time of compilation, the linker generates the function list according to information that the compiler has automatically extracted.
- When an object file is created without **-control_flow_integrity** specified at the time of compilation, the linker generates function lists for all symbols that were resolved for relocation in the object file.
- To add specific functions to the function list, specify the **-cfi_add_func** optimizing linkage editor (rlink) option.
  When a function in the specific object file is to be exempted from the function list, specify the **-cfi_ignore_module** optimizing linkage editor (rlink) option.

-cfi_add_func [V2.08.00 or later]

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-cfi_add_func={ <function symbol> | <function address> }[,{ <function symbol> | <func-
tion address> }]...
```

[Description]

- This option registers the symbol or address of functions in the function list for use in detecting illegal indirect function calls.
  For details on detecting illegal indirect function calls, refer to the item on the '-control_flow_integrity [Professional Edition only] [V2.08.00 or later]' compile option.
- Specify addresses in hexadecimal.
- If the specified symbol of a function is not included in the load module that was optimized by the linker, an error will occur.
- If this option is specified more than once, all specified symbols or addresses of functions are registered in the function list.
- When this option is used, the **-cfi** option must also be specified. If the **-cfi** option is not specified, an error will occur.

[Example]

- To register the sub1 function of the C source code, function address 0x100, and the function sub2 in the C source code in the function list, write this as:

```
-cfi_add_func=_sub1,100 -cfi_add_func=_sub2
```

-cfi_ignore_module [V2.08.00 or later]

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-cfi_ignore_module=<suboption>[,...]
                   <suboption>: <module> | <library file>[(<library module>
                    [,<library module>]...)[,...]]
```

[Description]

- This option specifies object files to be exempted from the function list for use in detecting illegal indirect function calls. For details on detecting illegal indirect function calls, refer to the item on the '-control_flow_integrity [Professional Edition only] [V2.08.00 or later]' compiler option.
- [V3.00.00 or later] This option specifies object files or library files to be exempted from the function list for use in detecting illegal indirect function calls. The module name can be used to specify a module in a library file.
- If the specified object file does not exist, an error will occur.
- If this option is specified more than once, the functions of all specified object files are exempted from the function list.
- When this option is used, the **-cfi** option must also be specified. If the **-cfi** option is not specified, an error will occur.

[Example]

- To remove functions in a.obj, b.obj, and the d module in c.lib from the function list, code as:

```
-cfi_ignore_module=a.obj,b.obj -cfi_ignore_module=c.lib(d)
```

-create_unfilled_area [V2.03.00 or later]

< Optimizing Linkage Editor (rlink) Options / Output Options >

[Format]

```
-create_unfilled_area
```

[Description]

- This option is available in V2.03 and later versions of this compiler.
- When a **Motorola S-record file** (**<name>.mot**) or **Hex file** (**<name>.hex**) is output, this option blocks spaces created by **.OFFSET** directives in the assembly language being filled with output data.
- When using this option, specify it when using the **ccrx** or **asrx** command to create an object file (**<name>.obj**) as well as when using the **rlink** command to create a **Motorola S-record file** or **Hex file**.

List Options

< Optimizing Linkage Editor (rlink) Options / List Options >

The following list options are available.

- -list
- -show

-list

< Optimizing Linkage Editor (rlink) Options / List Options >

[Format]

```
-list [=<file name>]
```

[Description]

- Specifies list file output and a list file name.
- If no list file name is specified, a list file with the same name as the output file (or first output file) is created, with the extension **lbp** when **form=library** or **extract** is specified, or **map** in other cases.
- Even if the section allocation address exceeds the allowable address range, this option outputs the link map information and symbol information. In this case, "**OVER**" is output. [V2.06.00 or later]

-show

< Optimizing Linkage Editor (rlink) Options / List Options >

[Format]

```
-show [=<sub>[,...]]
        <sub>:{ symbol | reference | section | xreference | total_size | vector |
                struct | relocation_attribute | all}
```

[Description]

- Specifies output contents of a list.
- Table 2.17 lists the suboptions.
- For details of list examples, refer to Linkage List, and Library List in the user's manual.

Table 2.17 Suboptions of show Option

No	Output Format	Suboption Name	Description
1	**form=library** is specified.	symbol	Outputs a symbol name list in a module (when **extract** is specified).
		reference	Not specifiable.
		section	Outputs a section list in a module.
		xreference	Not specifiable.
		total_size	Not specifiable.
		vector	Not specifiable.
		relocation_attribute	Not specifiable.
		cfi	Not specifiable.
		all	Not specifiable (when **extract** is specified). Outputs a symbol name list and a section list in a module (when **form=library**).

No	Output Format	Suboption Name	Description
2	Other than **form=library** is specified.	symbol	Outputs symbol address, size, type, and optimization contents.
		reference	Outputs the number of symbol references.
		section	Not specifiable.
		xreference	Outputs the cross-reference information.
		total_size	Shows the total sizes of sections allocated to the ROM and RAM areas.
		vector	Outputs vector information.
		struct	Outputs structure/union member information.
		relocation_attribute	When **-form=abs** is specified and **-strip** is not specified, the relocation attribute corresponding to the section is output. When **-form=hex/bin/stype** is specified and the input file format is not **absolute/hex/stype**, the relocation attribute corresponding to the section is output. Otherwise, **relocation_attribute** is not specified.
		cfi	Outputs the function list for use in detecting illegal indirect function calls. **cfi** must be specified with **-cfi** option (Otherwise an error will occur). **cfi** is specifiable when * **-form=abs** option is specified and **-strip** option is not specified, or * **-form=hex/bin/stype** option is specified and input files are **absolute/hex/stype**. Otherwise, **cfi** is not specifiable.
		all	If **form=rel** the linkage editor outputs the same information as when **show=symbol**,**xreference**,**total_size** is specified. If **form=rel,data_stuff** have been specified, the linkage editor outputs the same information as when **show=symbol,total_size** is specified. If **form=abs** the linkage editor outputs the same information as when **show=symbol**,**reference**,**xreference**,**total_size,struct** is specified. If **form=hex/stype/bin** the linkage editor outputs the same information as when **show=symbol**,**reference**,**xreference**,**total_size,struct** is specified. If **form=obj**, **all** is not specifiable.

[Remarks]

- The following table shows whether suboptions will be valid or invalid by all possible combinations of options **form**, **show**, and/or **show=all**.

		Symbol	Refer-ence	Section	Xrefer-ence	Vector	Total_size	relocation_attribute	cfi
form=abs	show	Valid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
	show=all	Valid	Valid	Invalid	Valid	Valid	Valid	Invalid	Invalid
form=lib	show	Valid	Invalid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid
	show=all	Valid	Invalid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid
form=rel	show	Valid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
	show=all	Valid	Invalid	Invalid	Valid[Note]	Invalid	Valid	Invalid	Invalid

		Symbol	Refer-ence	Section	Xrefer-ence	Vector	Total_size	relocation_attribute	cfi
form=obj	show	Valid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
	show=all	Valid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
form=hex/bin/sty	show	Valid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
	show=all	Valid	Valid	Invalid	Valid	Valid[Note]	Valid[Note]	Invalid	Invalid

Note The option is invalid if an absolute-format file is input.

- Note the following limitations on output of the cross-reference information.
  - When an absolute-format file is input, the referrer address information is not output.
  - Information about references to constant symbols within the same file is not output.
  - When optimization is specified at compilation, information about branches to immediate subordinate functions is not output.
  - When optimization of access to external variables is specified, information about references to variables other than base symbols is not output.
  - Both **show=total_size** and **total_size** output the same information.
  - When show=reference is valid, the number of references of the variable specified by #pragma address is output as 0.

Optimize Options

< Optimizing Linkage Editor (rlink) Options / Optimize Options >

The following optimize options are available.

- -optimize
- -nooptimize
- -samesize
- -symbol_forbid
- -samecode_forbid
- -section_forbid
- -absolute_forbid
- -ALLOW_OPTIMIZE_ENTRY_BLOCK [V3.06.00 or later]

-optimize

< Optimizing Linkage Editor (rlink) Options / Optimize Options >

[Format]

```
-optimize [= <suboption>[,...] ]
<suboption>: { SYmbol_delete | SAMe_code | SHort_format | Branch | SPeed  | SAFe }
```

- [Default]
  When this option is omitted, the default is **optimize**.

[Description]

- When **optimize** is specified, optimization is performed for the file specified with the **goptimize** option at compilation or assembly.
- -optimize (no suboptions) executes all optimization. It has the same meaning as
  -optimize=symbol_delete,same_code,short_format,branch.
- -optimize=speed executes optimizations other than those reducing object speed. It has the same meaning as
  -optimize=symbol_delete,short_format,branch
- -optimize=safe executes optimization other than those limited by variable or function attributes. It has the same meaning as -optimize=short_format,branch
- Other suboptions mean optimization as the following table.

Table 2.18 Suboptions of optimize Option

Suboption	Description	Program to be Optimized[Note1]	
		RXC	RXA
symbol_delete	Deletes variables/functions that are not referenced. Always be sure to specify **#pragma entry** at compilation or the **entry** option in the optimizing linkage editor.	O	X
same_code	Creates a subroutine for the same instruction sequence.	O	X
short_format	Replaces an instruction having a displacement or an immediate value with a smaller-size instruction when the code size of the displacement or immediate value can be reduced.	O	O
branch	Optimizes branch instruction size according to program allocation information. Even if this option is not specified, it is performed when any other optimization is executed.	O	O

Notes 1. RXC: C/C++ program for RX Family,
RXA: Assembly program for RX Family

[Remarks]

- When **form={object | relocate | library}** or **strip** is specified, this option is unavailable.
- **optimize=symbol_delete** is invalid unless the execution start address is specified by using either **#pragma entry** or the **entry** option.

-nooptimize

< Optimizing Linkage Editor (rlink) Options / Optimize Options >

[Format]

```
-nooptimize
```

- [Default]
  When this option is omitted, the default is **optimize**.

[Description]

- When **pnooptimize** is specified, optimization is not performed at linkage.

-samesize

< Optimizing Linkage Editor (rlink) Options / Optimize Options >

[Format]

```
-samesize = <size>
```

- [Default]
  When this option is omitted, the default is **samesize=1E**.

[Description]

- Specifies the minimum code size for the optimization with the same-code unification (**optimize=same_code**). Specify a hexadecimal value from 8 to 7FFF.

[Remarks]

- When **optimize=same_code** is not specified, this option is unavailable.

-symbol_forbid

< Optimizing Linkage Editor (rlink) Options / Optimize Options >

[Format]

```
-symbol_forbid = <symbol name> [,...]
```

[Description]

- Disables optimization regarding unreferenced symbol deletion. For a C/C++ variable or C function name, add an underscore (_) at the head of the definition name in the program. For a C++ function, enclose the definition name in the program with double-quotes including the parameter strings. When the parameter is **void**, specify as "<function name>( )".

[Remarks]

- If optimization is not applied at linkage, this option is ignored.

-samecode_forbid

< Optimizing Linkage Editor (rlink) Options / Optimize Options >

[Format]

```
-samecode_forbid = <function name> [,...]
```

[Description]

- Disables optimization regarding same-code unification. For a C/C++ variable or C function name, add an underscore (_) at the head of the definition name in the program. For a C++ function, enclose the definition name in the program with double-quotes including the parameter strings. When the parameter is **void**, specify as "<function name>( )".

[Remarks]

- If optimization is not applied at linkage, this option is ignored.

-section_forbid

< Optimizing Linkage Editor (rlink) Options / Optimize Options >

[Format]

```
-section_forbid  = <sub>[,...]
                   <sub>: [<file name>|<module name>](<section name>[,...])
```

[Description]

- Disables optimization for the specified section. If an input file name or library module name is also specified, the optimization can be disabled for a specific file, not only the entire section.

[Remarks]

- If optimization is not applied at linkage, this option is ignored.
- To disable optimization for an input file with its path name, type the path with the file name when specifying **section_forbid**.

-absolute_forbid

< Optimizing Linkage Editor (rlink) Options / Optimize Options >

[Format]

```
-absolute_forbid = <address> [+<size>] [,...]
```

[Description]

- Disables optimization regarding address + size specification.

[Remarks]

- If optimization is not applied at linkage, this option is ignored.

-ALLOW_OPTIMIZE_ENTRY_BLOCK [V3.06.00 or later]

< Optimizing Linkage Editor (rlink) Options / Optimize Options >

[Format]

```
-ALLOW_OPTIMIZE_ENTRY_BLOCK
```

- [Default]
  Optimization is not performed on any area allocated before the execution start symbol.

[Description]

- Performs optimization on the areas that are allocated before the execution start symbol.
- Specifying this option more than once has the same effect as specifying it once only. A warning is output in this case.

[Example]

To perform optimization including the areas that are allocated before the execution start symbol, describe as:

```
> rlink a.obj b.obj -optimize -entry=_main -allow_optimize_entry_block
```

[Remarks]

- This option is invalid for link processing that does not use optimization.
- If an address is specified by the **-entry** option, this option outputs a warning and ignores the specification.
- This option is invalid unless the **-entry** option is specified.

Section Options

< Optimizing Linkage Editor (rlink) Options / Section Options >

The following section options are available.

- -start
- -fsymbol
- -aligned_section

-start

< Optimizing Linkage Editor (rlink) Options / Section Options >

[Format]

```
-start = <sub> [,...]
         <sub>: [(] <section name> [{ : | , } <section name> [,...] ] [)] [,...] [ /
<address>]
```

- [Default]
  The section is allocated at 0.

[Description]

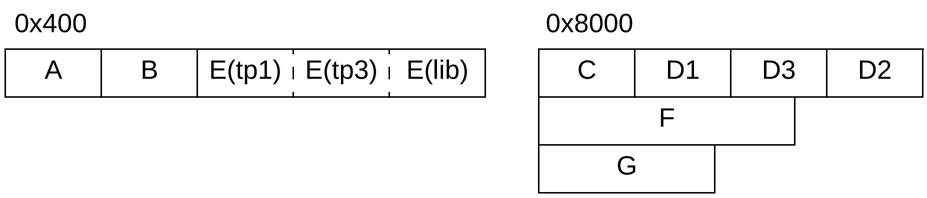
- Specifies the start address of the section. Specify an address as the hexadecimal.
- The section name can be specified with wildcards "*". Sections specified with wildcards are expanded according to the input order.
- Two or more sections can be allocated to the same address (i.e., sections are overlaid) by separating them with a colon ":".
- Sections specified at a single address are allocated in the specification order.
- Sections to be overlaid can be changed by enclosing them by parentheses "()".
- Objects in a single section are allocated in the specification order of the input file or the input library.
- If no address is specified, the section is allocated at 0.
- A section which is not specified with the **start** option is allocated after the last allocation address.

[Examples]

This example shows how sections are allocated when the objects are input in the following order (names enclosed by parentheses are sections in the objects).

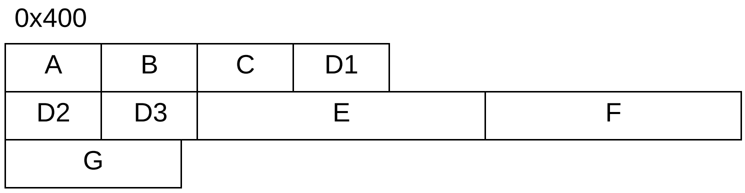
  - tp1.obj(A,D1,E) -> tp2.obj(B,D3,F)) -> tp3.obj(C,D2,E,G) -> lib.lib(E)
  -

- -start=A,B,E/400,C,D*:F:G/8000



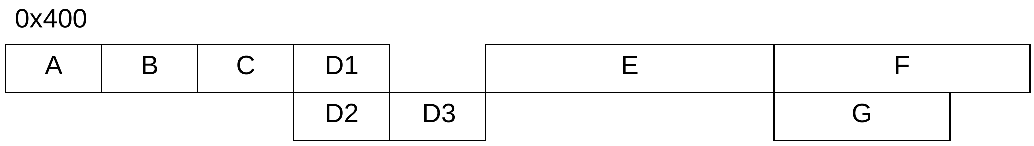
  - Sections C, F, and G separated by colons are allocated to the same address.
  - Sections specified with wildcards "*" (in this example, the sections whose names start with D) are allocated in the input order.
  - Objects in the sections having the same name (E in this example) are allocated in the input order.
  - An input library's section having the same name (E in this example) as those of input objects is allocated after the input objects.

- -start=A,B,C,D1:D2,D3,E,F:G/400



- The sections that come immediately after the colons (**A**, **D2**, and **G** in this example) are selected as the start and allocated to the same address.

- -start=A,B,C,(D1:D2,D3),E,(F:G)/400



- When the sections to be allocated to the same address are enclosed by parentheses, the sections within parentheses are allocated to the address immediately after the sections that come before the parentheses (C and E in this example).
- The section that comes after the parentheses (E in this example) is allocated after the last of the sections enclosed by the parentheses.

[Remarks]

- When **form={object | relocate | library}** or **strip** is specified, this option is unavailable.
- Parentheses cannot be nested.
- One or more colons must be written within parentheses. Parentheses cannot be written without a colon.
- Colons cannot be written outside of parentheses.
- When this option is specified with parentheses, optimization with the linkage editor is disabled.

-fsymbol

< Optimizing Linkage Editor (rlink) Options / Section Options >

[Format]

```
-fsymbol = <section name> [,...]
```

[Description]

- Outputs externally defined symbols in the specified section to a file in the assembler directive format.
- The file name is **<output file>.fsy**.

[Examples]

- Outputs externally defined symbols in sections **sct2** and **sct3** to **test.fsy**.

```
fsymbol = sct2, sct3
output=test.abs
```

- [Output example of **test.fsy**]

```
;RENESAS OPTIMIZING LINKER GENERATED FILE 2012.07.19
;fsymbol = sct2, sct3
;SECTION NAME = sct2
 .glb_f
_f: .equ 00000000h
 .glb_g
_g: .equ 00000016h
;SECTION NAME = sct3
 .glb _main
_main: .equ 00000020h
 .end
```

[Remarks]

- When **form={object | relocate | library}** or **strip** is specified, this option is unavailable.

-aligned_section

< Optimizing Linkage Editor (rlink) Options / Section Options >

[Format]

```
-aligned_section = <section name>[,...]
```

[Description]

- Changes the alignment value for the specified section to 16 bytes.

[Remarks]

- When **form={object | relocate | library}**, **extract**, or **strip** is specified, this option is unavailable.

Verify Options

< Optimizing Linkage Editor (rlink) Options / Verify Options >

The following verify options are available.

- -cpu
- -contiguous_section

-cpu

< Optimizing Linkage Editor (rlink) Options / Verify Options >

[Format]

```
-cpu={ <memory type> = <address range> [,...] | STRIDE}
       <memory type>:  { ROm | RAm | FIX }
       <address range>:  <start address> - <end address>
```

[Description]

- When **cpu=stride** is not specified, a section larger than the specified range of addresses leads to an error.
- When **cpu=stride** is specified, a section larger than the specified range of addresses is allocated to the next area of the same memory type or the section is divided.

[Examples]

- When the **stride** suboption is not specified:

```
start=D1,D2/100
cpu=ROM=100-1FF,RAM=200-2FF
```

  - The result is normal when **D1** and **D2** are respectively allocated within the ranges from 100 to 1FF and from 200 to 2FF. If they are not allocated within the ranges, an error will be output.

- When the **stride** suboption is specified:

```
start=D1,D2/100
cpu=ROM=100-1FF,RAM=200-2FF,ROM=300-3FF
cpu=stride
```

  - The result is normal when **D1** and **D2** are allocated within the ROM area (regardless of whether the section is divided). A linkage error occurs when they are not allocated within the ROM area even though the section is divided.
  - Specify an address range in which a section can be allocated in hexadecimal notation. The memory type attribute is used for the inter-module optimization.
  - **FIX** for <memory type> is used to specify a memory area where the addresses are fixed (e.g. I/O area).
  - If the address range of <start>-<end> specified for **FIX** overlaps with that specified for another memory type, the setting for **FIX** is valid.
  - When <memory type> is **ROM** or **RAM** and the section size is larger than the specified memory range, sub-option **STRIDE** can be used to divide a section and allocate them to another area of the same memory type. Sections are divided in module units.

```
cpu=ROM=0-FFFF,RAM=10000-1FFFF
```

  - Checks that section addresses are allocated within the range from 0 to FFFF or from 10000 to 1FFFF.
  - Object movement is not provided between different attributes with the inter-module optimization.

```
cpu=ROM=100-1FF,ROM=400-4FF,RAM=500-5FF
cpu=stride
```

  - When section addresses are not allocated within the range from 100 to 1FF, the linkage editor divides the sections in module units and allocates them to the range from 400 to 4FF.

[Remarks]

- When **form={object | relocate | library}** or **strip** is specified, this option is unavailable.
- When **cpu=stride** and **memory=low** are specified, this option is unavailable.
- When section **B** is divided by **cpu=stride**, the size of section **C$BSEC** increases by 8 bytes × number of divisions because this amount of information is required for initialization.

-contiguous_section

< Optimizing Linkage Editor (rlink) Options / Verify Options >

[Format]

```
-contiguous_section=<section name>[,...]
```

[Description]

- Allocates the specified section to another available area of the same memory type without dividing the section when **cpu=stride** is valid.

[Examples]

```
start=P,PA,PB/100
cpu=ROM=100-1FF,ROM=300-3FF,ROM=500-5FF
cpu=stride
contiguous_section=PA
```

- Section **P** is allocated to address 100.
- If section **PA** which is specified as **contiguous_section** is over address 1FF, section **PA** is allocated to address 300 without being divided.
- If section **PB** which is not specified as **contiguous_section** is over address 3FF, section **PB** is divided and allocated to address 500.

[Remarks]

- When **cpu=stride** is invalid, this option is unavailable.

Other Options

< Optimizing Linkage Editor (rlink) Options / Other Options >

The following other options are available.

- -s9
- -stack
- -compress
- -nocompress
- -memory
- -rename
- -lib_rename [V3.01.00 or later]
- -delete
- -replace
- -extract
- -strip
- -change_message
- -hide
- -total_size
- -verbose [V3.03.00 or later]

-s9

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-s9
```

[Description]

- Outputs the **S9** record at the end even if the entry address exceeds 0x10000.

[Remarks]

- When **form=stype** is not specified, this option is unavailable.

-stack

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-stack
```

[Description]

- Outputs a stack information file.
- The file name is **<output file name>.sni**.

[Remarks]

- When **form={object | relocate | library}** or **strip** is specified, this option is unavailable.

-compress

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-compress
```

- [Default]
  If this option is omitted, the debugging information is not compressed.

[Description]

- The debugging information is compressed.
- By compressing the debugging information, the debugger loading speed is improved.

[Remarks]

- When **form={object | relocate | library | hexadecimal | stype | binary}** or **strip** is specified, this option is unavailable.

-nocompress

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-nocompress
```

- [Default]
  If this option is omitted, the debugging information is not compressed.

[Description]

- The debugging information is not compressed.
- If the **nocompress** option is specified, the link time is reduced.

-memory

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-memory = [ High | Low ]
```

- [Default]
  The default for this option is **memory = high**.

[Description]

- Specifies the memory size occupied for linkage.
- When **memory = high** is specified, the processing is the same as usual.
- When **memory = low** is specified, the linkage editor loads the information necessary for linkage in smaller units to reduce the memory occupancy. This increases file accesses and processing becomes slower when the occupied memory size is less than the available memory capacity.
- **memory = low** is effective when processing is slow because a large project is linked and the memory size occupied by the linkage editor exceeds the available memory in the machine used.

[Remarks]

- When one of the following options is specified, the **memory=low** option is unavailable:
- When **form=absolute**, **hexadecimal**, **stype**, or **binary** is specified:
  compress, **delete**, **rename**, **map**, **stack**, **cpu=stride**, or
  list and **show[={reference | xreference}]** are specified in combination.
- When **form=library** is specified:
  delete, **rename**, **extract**, **hide**, **replace**, or **allow_duplicate_module_name**
- When **form=object** or **relocate** is specified:
  extract
- Some combinations of this option and the input or output file format are unavailable.

-rename

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-rename = <suboption> [,...]
          <suboption>: {[<file>] (<name> = <name> [,...])
                      | [<module>] (<name> = <name> [,...] ) }
```

[Description]

- Modifies an external symbol name or a section name.
- Symbol names or section names in a specific file or library in a module can be modified.
- For a C/C++ variable name, add an underscore (_) at the head of the definition name in the program.
- When a function name is modified, the operation is not guaranteed.
- If the specified name matches both section and symbol names, the symbol name is modified.
- If there are several files or modules of the same name, the priority depends on the input order.

[Examples]

```
rename=(_sym1=data)          ; Modifies _sym1 to data.
rename=lib1(P=P1)            ; Modifies the section P to P1
                             ; in the library module lib1.
```

[Remarks]

- When **extract** or **strip** is specified, this option is unavailable.
- When **form=absolute** is specified, the section name of the input library cannot be modified.
- Operation is not guaranteed if this option is used in combination with compile option **-merge_files**.

-lib_rename [V3.01.00 or later]

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-lib_rename = <name1>=<name2>[,...]
-lib_rename = <file>(<name1>=<name2>[,...])
-lib_rename = "<file>|<module>[|<module>...](<name1>=<name2>[,...])"
```

[Description]

- This option changes the name of global symbol or section included in a module within the library that was specified by the **-library** option.
- Specify the symbol name or section name to be changed as <name 1>. Specify the symbol name or section name after the change as <name 2>.
- When a C variable name is specified, add prefix "_" to the definition name in the program.
- If the specified name matches both section and symbol names, the symbol name is changed.
- If there are two or more files or modules with the same name, the priority depends on the input order.
- If this option is specified more than once, all specifications will be valid.
- An error will occur in any of the following cases.
    - When the specified <name>, <file>, or <module> cannot be found
    - When the parameter is omitted

[Examples]

- To change "_sym1" in b.lib and c.lib to "_data", describe as:

```
> rlink a.obj -lib=b.lib,c.lib  -lib_rename=(_sym1=_data)
```

- To change "_sym1" in all modules in b.lib to "_data", describe as:

```
> rlink a.obj -lib=b.lib,c.lib  -lib_rename=b.lib(_sym1=_data)
```

- To change "_sym1" in modules m1 and m2 in b.lib to "_data", describe as:

```
> rlink a.obj -lib=b.lib,c.lib  -lib_rename="b.lib|m1|m2(_sym1=_data)"
```

[Remarks]

- If this option is specified together with the **-form={object|library}**, **-extract**, or **-strip** option, an error will occur.
- When the **-form={absolute|hexadecimal|stype|binary}** option is specified, the **-show=struct** option cannot be specified together. Thus, the section name of the input library cannot be changed.
- Correct operation is not guaranteed if this option is used in combination with the compiler option **-merge_files**.

-delete

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-delete = <suboption> [,...]
          <suboption>: {[<file>] (<name>[,...]) | <module>}
```

[Description]

- Deletes an external symbol name or library module.
- Symbol names or modules in the specified file can be deleted.
- For a C/C++ variable name or C function name, add an underscore (_) at the head of the definition name in the program. For a C++ function name, enclose the definition name in the program with double-quotes including the parameter strings. If the parameter is **void**, specify as "<function name>()". If there are several files or modules of the same name, the file that is input first is applied.
- When a symbol is deleted using this option, the object is not deleted but the attribute is changed to the internal symbol.

[Examples]

```
delete=(_sym1)              ; Deletes the symbol _sym1 in all files.
delete=file1.obj(_sym2)     ; Deletes the symbol _sym2 in the file file1.obj.
```

[Remarks]

- When **extract** or **strip** is specified, this option is unavailable.
- When **form=library** has been specified, this option deletes modules.
- When **form={absolute|relocate|hexadecimal|stype|binary}** has been specified, this option deletes external symbols.
- Operation is not guaranteed if this option is used in combination with compile option **-merge_files**.

-replace

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-replace = <suboption> [,...]
           <suboption>: <file name> [ ( <module name> [,...] ) }
```

[Description]

- Replaces library modules.
- Replaces the specified file or library module with the module of the same name in the library specified with the **library** option.

[Examples]

```
replace=file1.obj           ; Replaces the module file1 with the module file1.obj.
replace=lib1.lib(mdl1)      ; Replaces the module mdl1 with the module mdl1
                            ; in the input library file lib1.lib.
```

[Remarks]

- When **form={object | relocate | absolute | hexadecimal | stype | binary}**, **extract**, or **strip** is specified, this option is unavailable.
- Operation is not guaranteed if this option is used in combination with compile option **-merge_files**.

-extract

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-extract = <module name> [,...]
```

[Description]

- Extracts library modules.
- Extracts the specified library module from the library file specified using the **library** option.

[Examples]

```
extract=file1           ; Extracts the module file1.
```

[Remarks]

- When **form={absolute | hexadecimal | stype | binary}** or **strip** is specified, this option is unavailable.
- When **form=library** has been specified, this option deletes modules.
- When **form={absolute|relocate|hexadecimal|stype|binary}** has been specified, this option deletes external symbols.

-strip

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-strip
```

[Description]

- Deletes debugging information in an absolute file or library file.
- When the **strip** option is specified, one input file should correspond to one output file.

[Examples]

```
input=file1.abs file2.abs file3.abs
strip
```

- Deletes debugging information of **file1.abs**, **file2.abs**, and **file3.abs**, and outputs this information to **file1.abs**, **file2.abs**, and **file3.abs**, respectively. Files before debugging information is deleted are backed up in **file1.abk**, **file2.abk**, and **file3.abk**.

[Remarks]

- When **form={object | relocate | hexadecimal | stype | binary}** is specified, this option is unavailable.

-change_message

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-change_message = <suboption> [,...]
                <suboption>: <error level> [= <error number> [-<error number>] [,...]
]
                 <error level>: {Information | Warning | Error}
```

[Description]

- Modifies the level of information, warning, and error messages.
- Specifies the execution continuation or abort at the message output.
- When a message number is specified, the error level of the message with the specified error number changes to the given level.
- A range of error message numbers can be specified by using a hyphen (-).
- Each error number must consist of a component number (05), phase (6), and a four-digit value (e.g. 2310 in the case of E0562310).
- If no error number is specified, all messages will be changed to the specified level.

[Examples]

```
change_message=warning=2310
```

- This changes E0562310 to a warning-level message so that linkage proceeds even if E0562310 is output.

```
change_message=error
```

- This changes all information and warning messages to error level messages.
  When a message is output, the execution is aborted.

-hide

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-hide
```

[Description]

- Deletes local symbol name information from the output file. Since all the name information regarding local symbols is deleted, local symbol names cannot be checked even if the file is opened with a binary editor. This option does not affect the operation of the generated file.
- Use this option to keep the local symbol names secret.
- The following types of symbol names are hidden:
- C source: Variable or function names specified with the **static** qualifiers
- C source: Label names for the **goto** statements
- Assembly source: Symbol names of which external definition (reference) symbols are not declared

Note The entry function name is not hidden.

[Examples]

- The following is a C source example in which this option is valid:

```
int g1;
int g2=1;
const int g3=3;
static int s1;          //<- The static variable name will be hidden.
static int s2=1;        //<- The static variable name will be hidden.
static const int s3=2;  //<- The static variable name will be hidden.

static int sub1()       //<- The static function name will be hidden.
{
    static int s1;      //<- The static variable name will be hidden.
    int l1;
    s1 = l1; l1 = s1;
    return(l1);
}

int main()
{
    sub1();
    if (g1==1)
        goto L1;
    g2=2;
L1:                     //<- The label name of the goto statement will be hidden.
    return(0);
}
```

[Remarks]

- This option is available only when the output file format is specified as **absolute**, **relocate**, or **library**.
- When the input file was compiled or assembled with the **goptimize** option specified, this option is unavailable if the output file format is specified as **relocate** or **library**.
- To use this option with the external variable access optimization, do not use this option for the first linkage, and use it only for the second linkage.
- The symbol names in the debugging information are not deleted by this option.

-total_size

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-total_size
```

[Description]

- Sends total sizes of sections after linkage to standard output. The sections are categorized as follows, with the overall size of each being output.
- Executable program sections
- Non-program sections allocated to the ROM area
- Sections allocated to the RAM area
- This option makes it easy to see the total sizes of sections allocated to the ROM and RAM areas.

[Remarks]

- The **show=total_size** option must be used if total sizes of sections are to be output in the linkage listing.
- When the ROM-image creation support (**rom** option) has been specified for a section, the section will be used by both the transfer source (ROM) and destination (RAM). The sizes of sections of this type will be added to the total sizes of sections in both ROM and RAM.

-verbose [V3.03.00 or later]

< Optimizing Linkage Editor (rlink) Options / Other Options >

[Format]

```
-verbose=<sub>[, ...]
sub : crc
```

[Description]

- This option displays the contents specified by the suboption in the standard error output.
- The suboption below can be specified.

CRC	This suboption displays the CRC operation result and its output address. Valid when the **crc** option is specified.

[Example]

- To display the CRC operation result and its output address in the standard error output, describe as:

```
> rlink a.obj -form=stype -start=.SEC1/1000 -crc=2000=1000-10ff/CCITT -verbose=crc
```

Subcommand File Option

< Optimizing Linkage Editor (rlink) Options / Subcommand File Option >

The following subcommand file option is available.

- -subcommand

-subcommand

< Optimizing Linkage Editor (rlink) Options / Subcommand File Option >

[Format]

```
-subcommand = <file name>
```

[Description]

- Specifies options with a subcommand file.
- The format of the subcommand file is as follows:
  <option> { = | Δ } [<suboption> [,...]] [Δ&] [;<comment>]
- The option and suboption are separated by an "=" sign or a space.
- For the **input** option, suboptions are separated by a space.
- One option is specified per line in the subcommand file.
- If a subcommand description exceeds one line, the description can be allowed to overflow to the next line by using an ampersand (&).

[Examples]

- Command line specification:

```
rlink file1.obj -sub=test.sub file4.obj
```

- Subcommand specification:

```
input   file2.obj file3.obj       ; This is a comment.
library lib1.lib, &               ; Specifies line continued.
lib2.lib
```

- Option contents specified with a subcommand file are expanded to the location at which the subcommand is specified on the command line and are executed.
- The order of file input is **file1.obj**, **file2.obj**, **file3.obj**, and **file4.obj**.

Options Other Than Above

< Optimizing Linkage Editor (rlink) Options / Options Other Than Above >

The following options other than above are available.

- -logo
- -nologo
- -end
- -exit

-logo

< Optimizing Linkage Editor (rlink) Options / Options Other Than Above >

[Format]

```
-logo
```

- [Default]
  When this option is omitted, the copyright notice is output.

[Description]

- The copyright notice is output.

-nologo

< Optimizing Linkage Editor (rlink) Options / Options Other Than Above >

[Format]

```
-nologo
```

- [Default]
  When this option is omitted, the copyright notice is output.

[Description]

- Output of the copyright notice is disabled.

-end

< Optimizing Linkage Editor (rlink) Options / Options Other Than Above >

[Format]

```
-end
```

[Description]

- Executes option strings specified before **END**. After the linkage processing is terminated, option strings that are specified after **END** are input and the linkage processing is continued.
- This option cannot be specified on the command line.

[Examples]

```
input=a.obj,b.obj        ; Processing (1)
start=P,C,D/100,B/8000   ; Processing (2)
output=a.abs             ; Processing (3)
end
input=a.abs              ; Processing (4)
form=stype               ; Processing (5)
output=a.mot             ; Processing (6)
```

- Executes the processing from (1) to (3) and outputs **a.abs**. Then executes the processing from (4) to (6) and outputs **a.mot**.

-exit

< Optimizing Linkage Editor (rlink) Options / Options Other Than Above >

[Format]

```
-exit
```

[Description]

- Specifies the end of the option specifications.
- This option cannot be specified on the command line.

[Examples]

- Command line specification:

```
rlink -sub=test.sub -nodebug
```

- test.sub:

```
input=a.obj,b.obj          ; Processing (1)
start=P,C,D/100,B/8000     ; Processing (2)
output=a.abs               ; Processing (3)
exit
```

- Executes the processing from (1) to (3) and outputs **a.abs**.
- The **nodebug** option specified on the command line after **exit** is executed is ignored.

2.5.4 Library Generator Options

The following shows the classification and description of options for the library generation phase.

Classification	Option	Description
Library Options	-head	Specifies a configuration library.
	-output	Specifies an output library file name.
	-nofloat	Creates a simple I/O function.
	-reent [V2.03.00 or later]	[To be supported by V2.03 and later versions] Creates a reentrant library.
	-lang	Selects the set of functions available from the C standard library.
	-simple_stdio	Creates a functionally cut down version of the set of I/O functions.
	-secure_malloc [Professional Edition only] [V2.05.00 or later]	Generates the calloc, free, malloc and realloc with security facility.
	-logo -nologo	Outputs the copyright. Disables output of the copyright.

Compile phase options can also be specified for the library generator. The specified options will be used for compile options when the library generator compiles a library. However, specification of some options might be ignored or might be fixed to specific values. For details about how the compile options specified for the library generator are interpreted, refer to Table 2.19.

Library Options

< Library Generator Options / Library Options >

The following library options are available.

- -head
- -output
- -nofloat
- -reent [V2.03.00 or later]
- -lang
- -simple_stdio
- -secure_malloc [Professional Edition only] [V2.05.00 or later]
- -logo
- -nologo

-head

< Library Generator Options / Library Options >

[Format]

```
-head=<sub>[,...]
<sub>:{ all | runtime | ctype | math | mathf | stdarg | stdio | stdlib | string | ios |
        new | complex | cppstring | c99_complex | fenv | inttypes | wchar | wctype}
```

- [Default]
  The default for this option is **head=all**.

[Description]

- This option specifies a configuration file with a header file name.
- When **head=all** is specified, all header file names will be configured.
- The runtime library is always configured.

-output

< Library Generator Options / Library Options >

[Format]

```
-output=<file name>
```

- [Default]
  The default for this option is **output=stdlib.lib**.

[Description]

- This option specifies an output file name.

-nofloat

< Library Generator Options / Library Options >

[Format]

```
-nofloat
```

[Description]

- This option creates simple I/O functions that do not support the conversion of floating-point numbers (%f, %e, %E, %g, %G).
- When inputting or outputting files that do not require the conversion of floating-point numbers, ROM can be saved.
  Target functions: fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, and vsprintf

[Remarks]

- In a library created with this option specified, correct operation cannot be guaranteed when floating-point numbers are input to or output from the target functions.

-reent [V2.03.00 or later]

< Library Generator Options / Library Options >

[Format]

```
-reent
```

[Description]

- This option creates reentrant libraries.
- Note that the rand, srand and EC++ library functions are not reentrant libraries.

[Remarks]

- This option is available in V2.03 and later versions of this compiler.
- When reentrant libraries are linked, use **#define** to define the macro name of **_REENTRANT** before including standard include files in the program or use the **-define** option to define **_REENTRANT** at compilation.

-lang

< Library Generator Options / Library Options >

[Format]

```
-lang = { c | c99 }
```

- [Default]
  The default for this option is **lang=c**.

[Description]

- This option selects which functions are to be usable in the C standard library.
- When **lang=c** is specified, only the functions conforming to the **C89** standard are included in the C standard library, and the extended functions of the **C99** standard are not included. When **lang=c99** is specified, the functions conforming to the **C89** standard and the functions conforming to the **C99** standard are included in the C standard library.

[Remarks]

- There are no changes in the functions included in the C++ and EC++ standard libraries.
- When **lang=c99** is specified, all functions including those specified by the **C99** standard can be used. Since the number of available functions is greater than when **lang=c** is specified, however, generating a library may take a long time.

-simple_stdio

< Library Generator Options / Library Options >

[Format]

```
-simple_stdio
```

[Description]

- This option creates a functional cutdown version of I/O functions.
- The functional cutdown version does not include the conversion of floating-point numbers (same as the function not supported with the **nofloat** option), the conversion of **long long** type, and the conversion of 2-byte code. When inputting or outputting files that do not require these functions, ROM can be saved.
  Target functions:fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, and vsprintf

[Remarks]

- In a library created with this option specified, correct operation cannot be guaranteed when a cutdown function is used in the target functions.
- This function is disabled during C++ and EC++ program compilation.

-secure_malloc [Professional Edition only] [V2.05.00 or later]

< Library Generator Options / Library Options >

[Format]

```
-secure_malloc
```

[Description]

This option creates the **calloc**, **free**, **malloc**, and **realloc** functions to which a security facility for detecting illegal operations to storage areas has been added.

When one of the following operations is performed, the **__heap_chk_fail** function is called.

- The pointer to an area other than that allocated by **calloc**, **malloc**, or **realloc** is passed to **free** or r**ealloc**.
- The pointer to an area released by **free** is passed again to **free** or **realloc**.
- A value is written to up to four bytes before and after the area allocated by calloc, malloc, or realloc and the pointer to that area is passed to **free** or **realloc**.

The same facility is also added to the **new** and **delete** operators in C++ programs.

The **__heap_chk_fail** function needs to be defined by the user and it describes the processing to be executed when an error occurs in management of dynamic memory.

Note the following points when defining the **__heap_chk_fail** function.

- The only possible type of return value is void and the **__heap_chk_fail** function does not have formal parameters.
- When defining the **__heap_chk_fail** function in a C++ program, add **extern "C"**.
- Corruption of heap space should not be detected recursively in the **__heap_chk_fail** function.
- Do not define the function as **static**.

[Example]

```
#include <stdlib.h>

void sub(int *ip) {
    ...
    free(ip);
}

int func(void) {
    int *ip;
    if ((ip = malloc(40 * sizeof(int))) == NULL)
        if ((ip = malloc(10 * sizeof(int))) == NULL) return(1);
        else sub(ip); /* First appearance of free */
    else
        ...
    free(ip); /* Second appearance of free */
    return(0);
}

#ifdef __cplusplus
extern "C" {
#endif
void __heap_chk_fail(void) {
    /* Processing when corruption of heap memory area is detected */
}
#ifdef __cplusplus
}
#endif
```

[Remarks]

The calloc, malloc, and realloc functions for the security facility allocate four extra bytes before and after each allocated area for the purpose of detecting writing to addresses outside the allocated area. This consumes more heap memory area than with the usual functions. Using the new operators in C++ programs will also consume more heap memory area.

-logo

< Library Generator Options / Library Options >

[Format]

```
-logo
```

- [Default]
  When this option is omitted, the copyright notice is output.

[Description]

- The copyright notice is output.

-nologo

< Library Generator Options / Library Options >

[Format]

```
-nologo
```

- [Default]
  When this option is omitted, the copyright notice is output.

[Description]

- Output of the copyright notice is disabled.

Compiler Options That Become Invalid

In addition to the options in 2.5.4 Library Generator Options, the C/C++ compiler options can be specified in the library generator as options used for library compilation. However, the options listed below are invalid; they are not selected at library compilation.

Table 2.19 Invalid Options

No.	Options that Become Invalid	Conditions for Invalidation	Library Configuration When Made Invalid
1	include	Always invalid	None
2	define	Always invalid	None
3	undefined	Always invalid	None
4	message nomessage	Always invalid	nomessage
5	change_message	Always invalid	None
6	file_inline_path	Always invalid	None
7	comment	Always invalid	None
8	check	Always invalid	None
9	output	Always invalid	output=obj
10	noline	Always invalid	None
11	debug nodebug	Always invalid	nodebug
12	listfile nolistfile show	Always invalid	nolistfile
13	file_inline	Always invalid	None
14	asmcmd	Always invalid	None
15	lnkcmd	Always invalid	None
16	asmopt	Always invalid	None
17	lnkopt	Always invalid	None
18	logo nologo	Always invalid	nologo
19	euc sjis latin1 utf8	Always invalid	None
20	outcode	Always invalid	None
21	subcommand	Always invalid	None
22	alias	Always invalid	alias=noansi
23	pic pid	lang=cpp or at C++ source compilation[Note1]	None
24	ip_optimize	Always invalid	None

No.	Options that Become Invalid	Conditions for Invalidation	Library Configuration When Made Invalid
25	merge_files	Always invalid	None
26	whole_program	Always invalid	None
27	big5 gb2312	Always invalid Note2	None
28	map [V3.02.00 or later]	Always invalid Note3	smap
29	control_flow_integrity	Always invalid	None
30	create_unfilled_area [V3.00.00 or later]	Always invalid	None
31	stack_protector [V3.00.00 or later]	Always invalid	None
32	stack_protector_all [V3.00.00 or later]	Always invalid	None
33	misra2004	Always invalid	None
34	misra2012 [V3.00.00 or later]	Always invalid	None
35	misra_intermodule [V3.01.00 or later]	Always invalid	None
36	tfu [V3.01.00 or later]	Always invalid	None
37	truncated_address_initializer [V3.01.00 or later]	Always invalid	None
38	g_line [V3.02.00 or later]	Always invalid	None
39	tfu_version [V3.05.00 or later]	Always invalid	None
40	nosave_tfu [V3.05.00 or later]	Always invalid	None

Notes 1. Warning W0511171 is output.

Notes 2. Error F0593305 is output. (This library cannot be generated.)

Notes 3. Any specification of <file name> is ignored. Even if <file name> does not exist, no error is output.

3. OUTPUT FILES

This chapter describes the format and other aspects of files output by a build via each command.

3.1 Assemble List File

This section covers the contents and format of the assemble list file output by the assembler.
The source list file contains the compilation and assembly results. Table 3.1 shows the structure and contents of the source list.

Table 3.1 Structure and Contents of Source List

No	Output Information	Contents	Suboption[Note]	When -show Option is not Specified
1	Source information	C/C++ source code corresponding to assembly source code	-show=source	Not output
2	Object information	Machine code used in object programs and the assembly source code	None	Output
3	Statistics information	Total number of errors, number of source program lines, and size of each section	None	Output
4	Command specification information	File names and options specified by the command	None	Output

Note Valid when the -listfile option is specified.

3.1.1 Source Information

The source information is included in the object information when the -show**=source** option is specified. For an example of source information, refer to the next section, Object Information.

3.1.2 Object Information

The following figure shows an example of object information output.

```
* RX FAMILY ASSEMBLER V2.00.00  [15 Feb 2013] *  SOURCE LIST  Mon Feb 18 20:15:19 2013
(1)      (2)                (3)    (4)
LOC.     OBJ.               0XMDA SOURCE STATEMENT

                                  ;RX Family C/C++ Compiler (V2.00.00  [15 Feb 2013])
18-Feb-2013 20:15:19

                                  ;*** CPU TYPE ***

                                  ;-ISA=RXV1

                                  ;*** COMMAND PARAMETER ***

                                  ;-output=src=sample.src
                                  ;-listfile
                                  ;-show=source
                                  ;sample.c

                                  .glb_x
```

```
                                  .glb_func02
                                  .glb_func03
                                  .glb_func01
                                  (5)      (6)
                                  ;LineNo. C-SOURCE STATEMENT

                                  .SECTIONP,CODE
00000000                          _func02:
                                  .STACK_func02=12
                                  ;        1 #include "include.h"
                                  ;        2 int func01(int);
                                  ;        3 int func03(int);
                                  ;        4
                                  ;        5 int func02(int z)
00000000 6E67                     PUSHM R6-R7
00000002 EF16                     MOV.L R1, R6
                                  ;        6 {
                                  ;        7     x = func01(z);
00000004 05rrrrrr              A  BSR _func01
00000008 FB72rrrrrrrr             MOV.L #_x, R7
0000000E E371                     MOV.L R1, [R7]
                                  ;        8     if (z == 2) {
00000010 6126                     CMP #02H, R6
00000012 18                    S  BNE L12
00000013                          L11:; bb3
                                  ;        9         x++;
00000013 6211                     ADD #01H, R1
00000015 08                    S  BRA L13
00000016                          L12:; bb6
                                  ;       10     } else {
                                  ;       11         x = func03(x + 2);
00000016 6221                     ADD #02H, R1
00000018 39rrrr                W  BSR _func03
0000001B                          L13:; bb13
0000001B E371                     MOV.L R1, [R7]
                                  ;       12     }
                                  ;       13     return x;
                                  ;       14 }
0000001D 3F6702                   RTSD #08H, R6-R7
00000020                          _func03:
                                  .STACK_func03=4
                                  ;       15
                                  ;       16 int func03(int p)
                                  ;       17 {
                                  ;       18     return p+1;
00000020 6211                     ADD #01H, R1
                                  ;       19 }
00000022 02                       RTS
                                  .SECTIOND,ROMDATA,ALIGN=4
00000000                          _y:
00000000 01000000                 .lword00000001H
                                    .END
```

Item Number	Description
(1)	Location information (LOC.) Location address of the object code that can be determined at assembly.
(2)	Object code information (OBJ.) Object code corresponding to the mnemonic of the source code.

<table>
<tr><th>Item Number</th><th colspan="6">Description</th></tr>
<tr><td rowspan="11">(3)</td><td colspan="6">Line information (0XMDA)
Results of source code processing by the assembler. The following shows the meaning of each symbol.</td></tr>
<tr><td>0</td><td>X</td><td>M</td><td>D</td><td>A</td><td>Description</td></tr>
<tr><td>0-30</td><td></td><td></td><td></td><td></td><td>Shows the nesting level of include files.</td></tr>
<tr><td></td><td>X</td><td></td><td></td><td></td><td>Shows the line where the condition is false in conditional assembly when -show=conditionals is specified.</td></tr>
<tr><td></td><td></td><td>M</td><td></td><td></td><td>Shows the line expanded from a macro instruction when -show=expansions is specified.</td></tr>
<tr><td></td><td></td><td>D</td><td></td><td></td><td>Shows the line that defines a macro instruction when -show=definitions is specified.</td></tr>
<tr><td></td><td></td><td></td><td>S</td><td></td><td>Shows that branch distance specifier S is selected.</td></tr>
<tr><td></td><td></td><td></td><td>B</td><td></td><td>Shows that branch distance specifier B is selected.</td></tr>
<tr><td></td><td></td><td></td><td>W</td><td></td><td>Shows that branch distance specifier W is selected.</td></tr>
<tr><td></td><td></td><td></td><td>A</td><td></td><td>Shows that branch distance specifier A is selected.</td></tr>
<tr><td></td><td></td><td></td><td></td><td>*</td><td>Shows that a substitute instruction is selected for a conditional branch instruction.</td></tr>
<tr><td>(4)</td><td colspan="6">Source information (SOURCE STATEMENT)
Contents of the assembly-language source file.</td></tr>
<tr><td>(5)</td><td colspan="6">C/C++ source line number (LineNo.)</td></tr>
<tr><td>(6)</td><td colspan="6">C/C++ source statement (C-SOURCE STATEMENT)
C/C++ source statement output when the -show=source option is specified.</td></tr>
</table>

3.1.3 Statistics Information

The following figure shows an example of statistics information output.

```
Information List (1)
TOTAL ERROR(S)    00000
TOTAL WARNING(S)  00000
TOTAL LINE(S)     00071  LINES
Section List (2)
Attr         Size               Name
CODE     0000000047(0000002FH)  P
ROMDATA  0000000004(00000004H)  D
```

Item Number	Description
(1)	Numbers of error messages and warning messages, and total number of source lines
(2)	Section information (section attribute, size, and section name)

3.1.4 Compiler Command Specification Information

The file names and options specified on the command line when the compiler is invoked are output. The compiler command specification information is output at the beginning of the list file. The following figure shows an example of command specification information output.

```
;*** CPU TYPE *** (1)
;-ISA=RXV1
;*** COMMAND PARAMETER *** (2)
;-output=src=C:\tmp\elp1894\sample.src
;-nologo
;-show=source
;sample.c
```

Item Number	Description
(1)	Selected microcomputer
(2)	File names and options specified for the compiler

3.1.5 Assembler Command Specification Information

The file names and options specified on the command line when the assembler is invoked are output. The assembler command specification information is output at the end of the list file. The following figure shows an example of command specification information output.

```
Cpu Type  (1)
-ISA=RXV1
Command Parameter  (2)
-output=sample.obj
-nologo
-listfile=sample.lst
```

Item Number	Description
(1)	Microcomputer selected for the assembler
(2)	File names and options specified for the assembler

3.2 Link Map File

This section explains the link map file.

The link map has information of the link result. It can be referenced for information such as the section's allocation addresses.

3.2.1 Structure of Linkage List

Table 3.2 shows the structure and contents of the linkage list.

Table 3.2 Structure and Contents of Linkage List

No	Output Information	Contents	When -show Option[Note] is Specified	When -show Option is not Specified
1	Option information	Option strings specified by a command line or subcommand	None	Output

No	Output Information	Contents	When -show Option-Note is Specified	When -show Option is not Specified
2	Error information	Error messages	None	Output
3	Linkage map information	Section name, start/end addresses, size, and type	None	Output
		When -show=relocation_attribute is specified, the relocation attribute is output.	-show=relocation_attribute	Not output
4	Symbol information	Static definition symbol name, address, size, and type in the order of address When -show=reference is specified: Symbol reference count and optimization information in addition to the above information When -show=struct is specified, information on the structure and union members is output.	-show =symbol -show =reference	Not output Not output
5	Symbol deletion optimization information	Symbols deleted by optimization	-show =symbol	Not output
6	Cross-reference information	Symbol reference information	-show =xreference	Not output
7	Total section size	Total sizes of RAM, ROM, and program sections	-show=total_size	Not output
8	Vector information	Vector numbers and address information	-show=vector	Not output
9	CRC information	CRC calculation result and output addresses	None	Always output when the CRC option is specified
10	CFI information	Contents of the function list for use in detecting illegal indirect function calls	-show=cfi	Not output

Note The -show option is valid when the **list** option is specified.

3.2.2 Option Information

The option strings specified by a command line or a subcommand file are output. The following figure shows an example of option information output when **rlink** -subcommand**=test.sub** -list -show is specified.

```
(test.sub contents)
INPUT test .obj
```

```
*** Options ***
-sub=test.sub       (1)
INPUT test .obj     (2)
-list               (1)
-show               (1)
```

Item Number	Description
(1)	Outputs option strings specified by a command line or a subcommand in the specified order.
(2)	Subcommand in the **test.sub** subcommand file.

3.2.3 Error Information

Error messages are output. The following figure shows an example of error information output.

```
*** Error Information ***
** E0562310 (E) Undefined external symbol "strcmp" referred to in "test.obj"   (1)
```

Item Number	Description
(1)	Outputs an error message.

Note As the number for the alignment of code sections whose address is determined after linkage, when **big endian** is selected, a multiple of 4 is indicated regardless of the actual number for alignment at the time of compiling and assembling.

3.2.4 Linkage Map Information

The start and end addresses, size, and type of each section are output in the order of address. The following figure shows an example of linkage map information output.

```
*** Mapping List ***
(1)                                (2)        (3)         (4)    (5)      (6)
SECTION                            START      END         SIZE   ALIGN    ATTRIBUTE
P
                                  00001000   00001000        1   1        CODE
C
                                  00001004   00001007        4   4        ROMDATA
D_2
                                  00001008   000014dd      4d6   2        ROMDATA
B_2
                                  000014de   000050b3     3bd6   2        DATA
```

Item Number	Description
(1)	Section name
(2)	Start address **OVER** being displayed here indicates that the address exceeded the range that can be expressed by 32 bits.
(3)	End address **OVER** being displayed here indicates that the address exceeded the range that can be expressed by 32 bits.
(4)	Section size
(5)	Section boundary alignment value
(6)	The relocation attribute is output

Note As the number for the alignment of code sections whose address is determined after linkage, when big endian is selected, a multiple of 4 is indicated regardless of the actual number for alignment at the time of compiling and assembling.

3.2.5 Symbol Information

When -show**=symbol** is specified, the addresses, sizes, and types of externally defined symbols or static internally defined symbols are output in the order of address. When -show**=reference** is specified, the symbol reference counts and optimization information are also output. The following figure shows an example of symbol information output.

```
*** Symbol List ***
SECTION=(1)
(2)          (3)          (4)           (5)
FILE=        START        END           SIZE
  (6)           (7)          (8)           (9)                  (10)     (11)
  SYMBOL        ADDR         SIZE          INFO                 COUNTS   OPT
SECTION=P
FILE=test.obj
                00000000     00000428           428
  _main
                00000000            2      func ,g                 0
  _malloc
                00000000           32      func ,l                 0
FILE=mvn3
                00000428     00000490            68
  $MVN#3
                00000428            0      none ,g                 0
```

Item Number	Description
(1)	Section name
(2)	File name
(3)	Start address of a section included in the file indicated by (2) above
(4)	End address of a section included in the file indicated by (2) above
(5)	Section size of a section included in the file indicated by (2) above
(6)	Symbol name
(7)	Symbol address **OVER** being displayed here indicates that the address exceeded the range that can be expressed by 32 bits.
(8)	Symbol size
(9)	Symbol type as shown below Data type: func: Function name data: Variable name entry: Entry function name none:Undefined (label, assembler symbol) Declaration type g: External definition l: Internal definition
(10)	Symbol reference count only when -show=reference is specified. * is output when show=reference is not specified.

Item Number	Description
(11)	Optimization information as shown below. ch: Symbol modified by optimization cr: Symbol created by optimization mv: Symbol moved by optimization

When the -show=struct option is specified, the addresses for the structure and union members that are defined in the source file for which the -debug option was specified at compilation are output. The output example of the symbol information is shown below.

```
*** Symbol List ***

SECTION=
FILE=                                 START        END    SIZE
  SYMBOL                              ADDR         SIZE   INFO       COUNTS   OPT
    (1)                                            (2)
    STRUCT                                         SIZE
      (3)                             (4)          (5)    (6)
      MEMBER                          ADDR         SIZE   INFO

SECTION=B
FILE=tp.obj
                                    00000000   0000000b         c
  _st
                                    00000000          c   data ,g         0
    struct  {
                                                      c
      _st.mem1
                                    00000000          1   char
      _st.mem2
                                    00000004          4   int
      _st.mem3
                                    00000008          2   short
    }
```

Number	Desciption
(1)	struct is output for a structure or union is output for a union.
(2)	Total size of the structure or union.
(3)	The member name is concatenated after a symbol name with a dot (.).
(4)	The member address is output.
(5)	The member size is output.
(6)	The member type is output.

3.2.6 Symbol Deletion Optimization Information

The size and type of symbols deleted by symbol deletion optimization (-optimize**=symbol_delete**) are output.
The following figure shows an example of symbol deletion optimization information output.

```
*** Delete Symbols ***
(1)                            (2)    (3)
SYMBOL                         SIZE   INFO
  _Version
                                  4   data ,g
```

Item Number	Description
(1)	Deleted symbol name
(2)	Deleted symbol size
(3)	Deleted symbol type as shown below Data type func: Function name data: Variable name Declaration type g: External definition l: Internal definition

3.2.7 Cross-Reference Information

The symbol reference information (cross-reference information) is output when -show=**xreference** is specified. The following figure shows an example of cross-reference information output.

```
*** Cross Reference List ***
(1)  (2)         (3)             (4)          (5)
No   Unit Name   Global.Symbol   Location     External Information
0001 a
     SECTION=P   _func
                                 00000100
                 _func1
                                 00000116
                 _main
                                 0000012c
                 _g
                                 00000136
     SECTION=B
                 _a
                                 00000190   0001(00000140:P)
                                            0002(00000178:P)
                                            0003(0000018c:P)
0002 b
     SECTION=P
                 _func01
                                 00000154   0001(00000148:P)
                 _func02
                                 00000166   0001(00000150:P)
0003 c
     SECTION=P
                 _func03
                                 00000184
```

Item Number	Description
(1)	Unit number, which is an identification number in object units
(2)	Object name, which specifies the input order at linkage
(3)	Symbol name output in ascending order of allocation addresses for every section

Item Number	Description
(4)	Symbol allocation address, which is a relative value from the beginning of the section when -form=relocate is specified **OVER** being displayed here indicates that the address exceeded the range that can be expressed by 32 bits.
(5)	Address of an external symbol that has been referenced Output format: <Unit number> (<address or offset in section>:<section name>)

3.2.8 Total Section Size

The total sizes of ROM, RAM, and program sections are output. The following figure shows an example of total section size output.

```
*** Total Section Size ***
RAMDATA SECTION :       00000660 Byte(s)  (1)
ROMDATA SECTION :       00000174 Byte(s)  (2)
PROGRAM SECTION :       000016d6 Byte(s)  (3)
```

Item Number	Description
(1)	Total size of RAM data sections
(2)	Total size of ROM data sections
(3)	Total size of program sections

3.2.9 Vector Information

The contents of the variable vector table are output when -show**=vector** is specified. The following figure shows an example of vector information output.

```
*** Variable Vector Table List ***
(1)     (2)
NO.     SYMBOL/ADDRESS
 0      $fdummy
 1      $fa
 2      00ff8800
 3      $fdummy
   :
<Omitted>
```

Item Number	Description
(1)	Vector number
(2)	Symbol. When no symbol is defined for the vector number, the address is output.

3.2.10 CRC Information

The CRC calculation result and output address are output when the CRC option is specified.

```
*** CRC Code ***
CODE    : cb0b      (1)
ADDRESS : 00007ffe  (2)
```

Item Number	Description
(1)	CRC calculation result
(2)	Address where the CRC calculation result is output

3.2.11 CFI Information

If -show=**cfi** is specified, this option outputs the contents of the function list for use in detecting illegal indirect function calls. The output example is given below.

```
*** CFI Function List ***

SYMBOL/ADDRESS

_func     (1)
0000F100  (2)
```

Item Number	Description
(1)	Outputs the symbol for the function.
(2)	Outputs the address of the function if a symbol for it has not been defined.

3.3 Library List

This section covers the contents and format of the library list output by the optimizing linkage editor.

3.3.1 Structure of Library List

Table 3.3 shows the structure and contents of the library list.

Table 3.3 Structure and Contents of Library List

No	Output Information	Contents	Suboption[Note]	When -show Option is not Specified
1	Option information	Option strings specified by a command line or subcommand	-	Output
2	Error information	Error messages	-	Output
3	Library information	Library information	-	Output
4	Information of modules, sections, and symbols within library	Module within the library	-	Output
		When **show=symbol** is specified: List of symbol names in a module within the library	-show=symbol	Not output
		When **show=section** is specified: Lists of section names and symbol names in a module within the library	-show=section	Not output

Note All options are valid when the -list option is specified.

3.3.2 Option Information

The option strings specified by a command line or a subcommand file are output.

The following figure shows an example of option information output when **rlink** -subcommand **= test.sub** -list -show is specified.

```
(test.sub contents)
form    library
in      adhry.obj
output  test.lib
```

```
*** Options ***
-sub=test.sub
form    library
in      adhry.obj
output  test.lib
-list
-show
```

(1)
(2)

Item Number	Description
(1)	Outputs option strings specified by a command line or a subcommand in the specified order.
(2)	Subcommand in the test.sub subcommand file.

3.3.3 Error Information

Messages for errors or warnings are output.
The following figure shows an example of error information output.

```
*** Error Information ***
** W0561200 (W) Backed up file "main.lib" into "main.lbk"  (1)
```

Item Number	Description
(1)	Outputs a warning message.

3.3.4 Library Information

The library type is output.
The following figure shows an example of library information output.

```
*** Library Information ***
LIBRARY NAME =test.lib  (1)
CPU=RX610               (2)
ENDIAN=Big              (3)
ATTRIBUTE=system        (4)
NUMBER OF MODULE =1     (5)
```

Item Number	Description
(1)	Library name
(2)	CPU name
(3)	Endian type
(4)	Library file attribute: either system library or user library
(5)	Number of modules within the library

3.3.5 Module, Section, and Symbol Information within Library

A list of modules within the library is output.
When -show**=symbol** is specified, the symbol names in a module within the library are listed. When -show**=section** is specified, the section names and symbol names in a module within the library are listed.
The following figure shows an output example of module, section, and symbol information within a library.

```
*** Library List ***
(1)             (2)
MODULE          LAST UPDATE
  (3)
  SECTION
   (4)
   SYMBOL
adhry
                29-Feb-2000 12:34:56
  P
   _main
   _Proc0
   _Proc1
  C
  D
   _Version
  B
   _IntGlob
   _CharGlob
```

Item Num-ber	Description
(1)	Module name
(2)	Module definition date If the module is updated, the latest module update date is displayed.
(3)	Section name within a module
(4)	Symbol within a section

3.4 S-Type and HEX File Formats

This section describes the S-type files and HEX files that are output by the optimizing linkage editor.

3.4.1 S-Type File Format

Figure 3.1 S-Type File Format

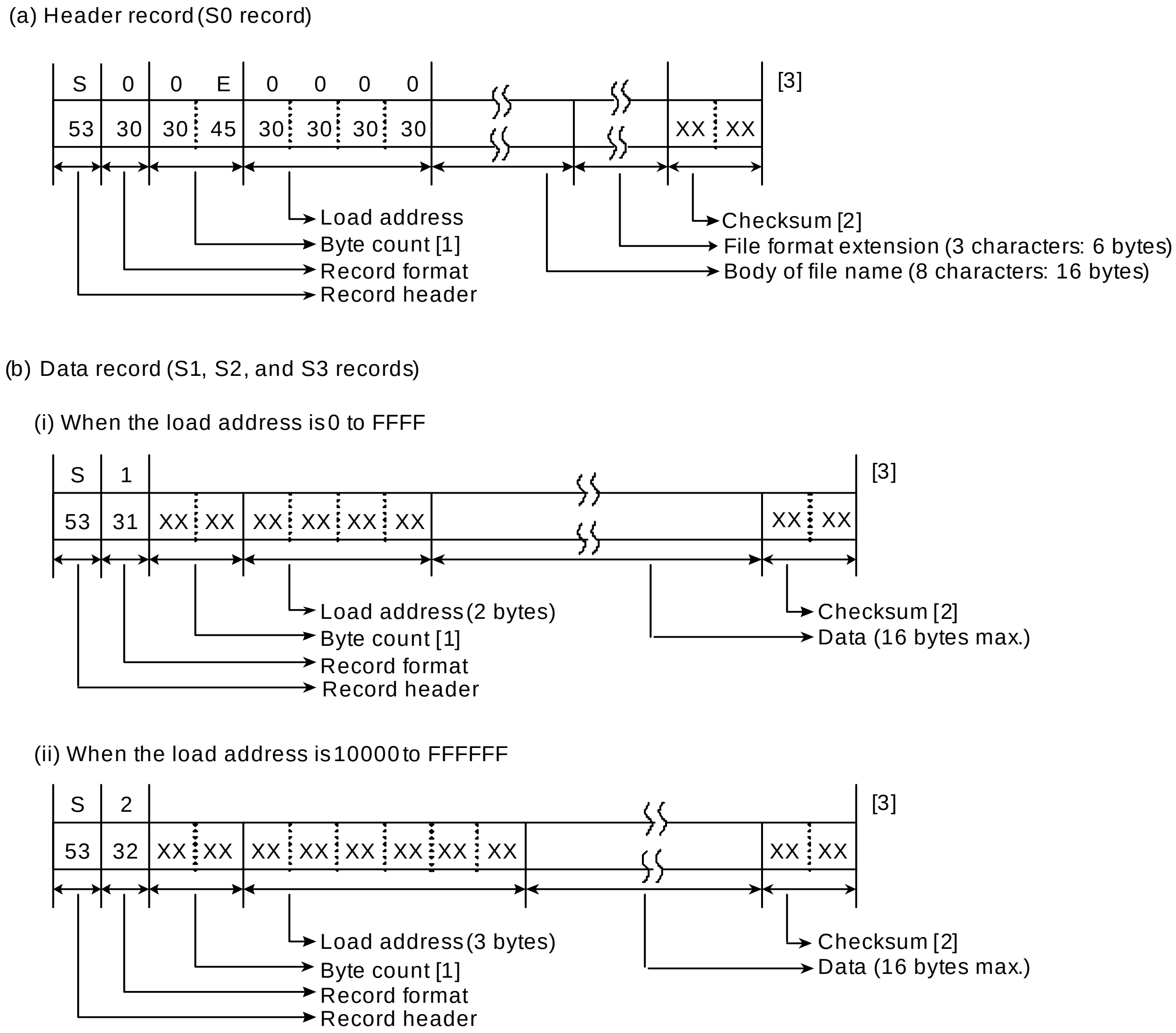
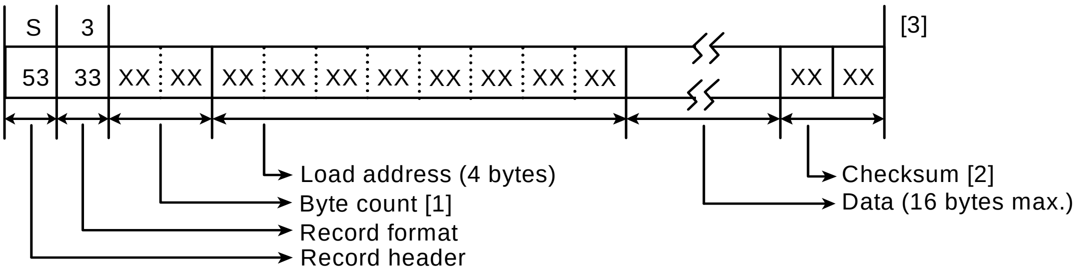


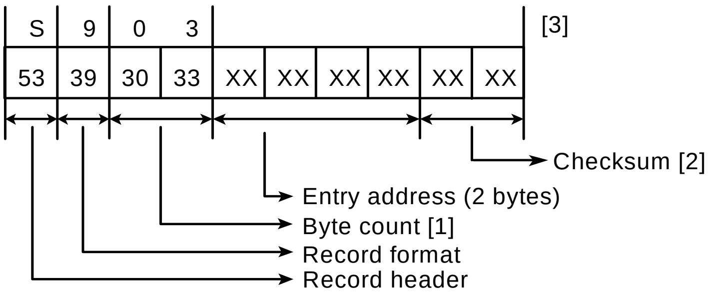
Figure 3.2 S-Type File Format (cont)

(iii) When the load address is 1000000 to FFFFFFFF

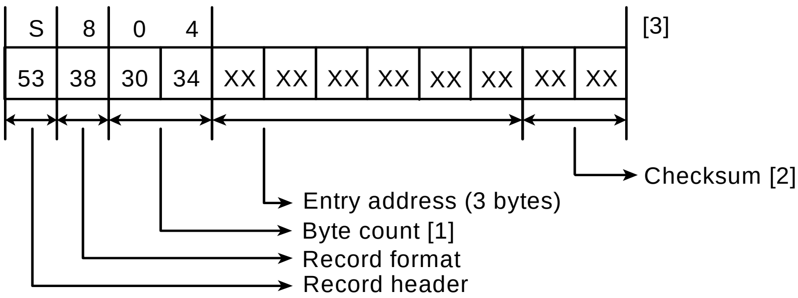


(c) End record (S9, S8, and S7 records)

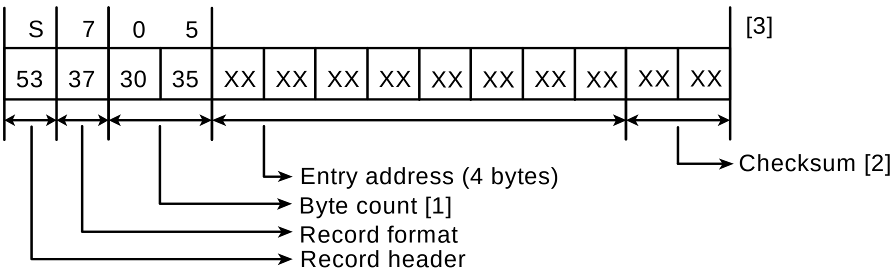
(i) When the entry address is 0 to FFFF



(ii) When the entry address is 10000 to FFFFFF



(iii) When the entry address is 1000000 to FFFFFFFF



Notes: [1] The number of bytes from the load address (or the entry address) to the checksum.

[2] 1's complement of the sum of the byte count and the data between the checksum and the byte count, in byte units.

[3] A new-line character is added immediately after the checksum.

3.4.2 HEX File Format

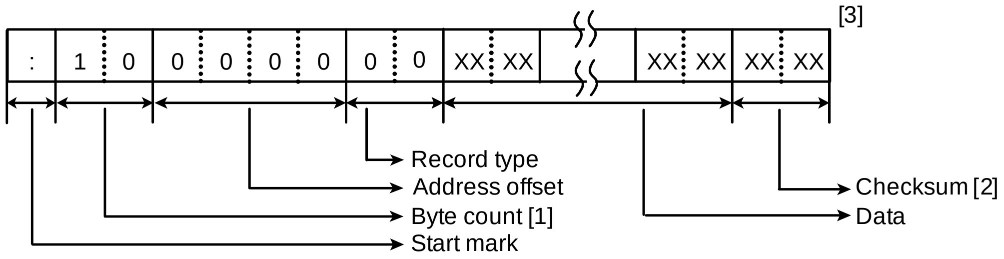
The execution address of each data record is obtained as described below.

(1) Segment address
(Segment base address << 4) + (Address offset of the data record)

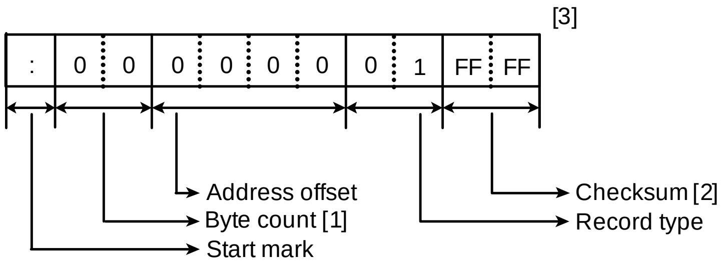
(2) Linear address
(Linear base address << 16) + (Address offset of the data record)

Figure 3.3 HEX File Format

(a) Data record (00 record)



(b) End record (01 record)



(c) Expansion segment address record(02 record)

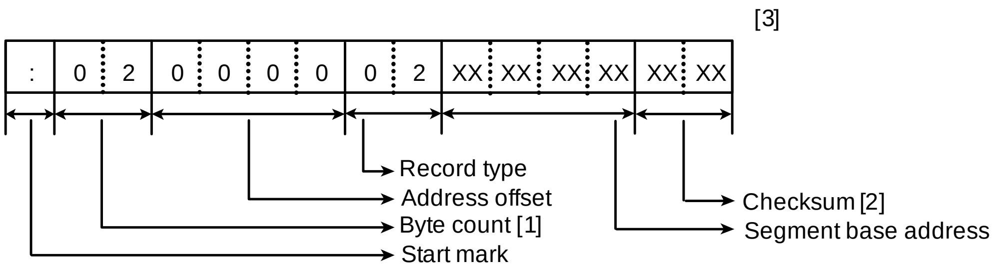
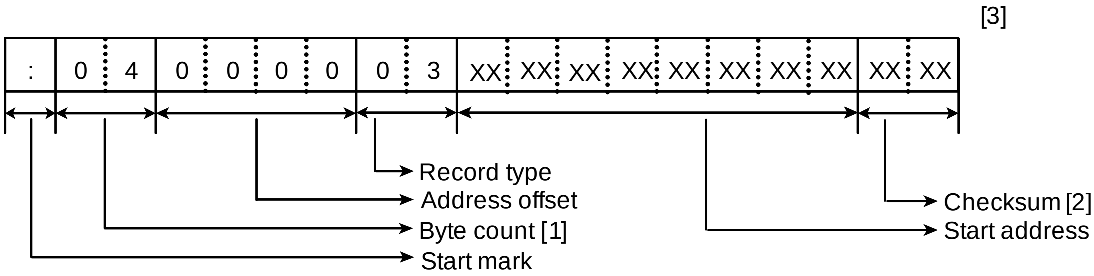
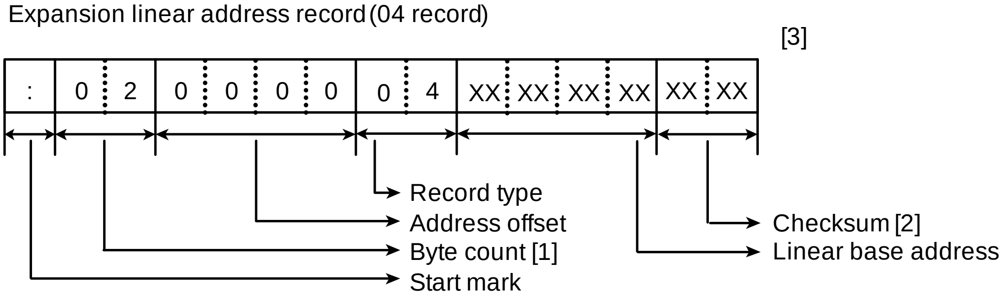


Figure 3.4 HEX File Format (cont)

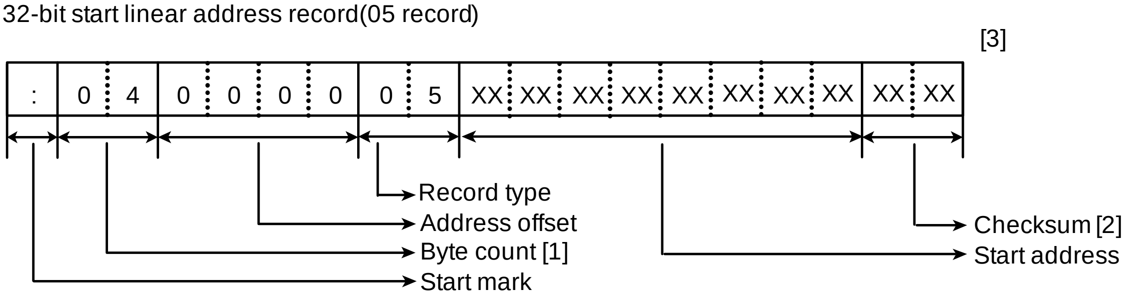
(d) Start address record (03 record)



(e) Expansion linear address record (04 record)



(f) 32-bit start linear address record(05 record)



Notes: [1] The number of bytes from the byte following the record type to the previous byte of the checksum.

[2] 2's complement of the sum of the byte count and the data between the byte count and checksum, in hexadecimal(lower 8 bits are valid).

[3] A new-line character is added immediately after the checksum.

4. COMPILER LANGUAGE SPECIFICATIONS

4.1 Basic Language Specifications

The RXC supports the language specifications stipulated by the ANSI standards. These specifications include items that are stipulated as processing definitions. This chapter explains the language specifications of the items dependent on the processing system of the RX microcontrollers.

For extended language specifications explicitly added by the RXC, refer to section 4.2 Extended Language Specifications.

4.1.1 Unspecified Behavior

This section describes behavior that is not specified by the ANSI standard.

(1) Execution environment - initialization of static storage
Static data is output during compilation as a data section.

(2) Meanings of character displays - backspace (\b), horizontal tab (\t), vertical tab (\t)
This is dependent on the design of the display device.

(3) Types - floating point
Conforms to IEEE754*.

Note IEEE: Institute of Electrical and Electronics Engineers
IEEE754 is a system for handling floating-point calculations, providing a uniform standard for data formats, numerical ranges, and the like handled.

(4) Expressions - evaluation order
In general, expressions are evaluated from left to right. The behavior when optimization has been applied, however, is undefined. Options or other settings could change the order of evaluation, so please do not code expressions with side effects.

(5) Function calls - parameter evaluation order
In general, function arguments are evaluated from first to last. The behavior when optimization has been applied, however, is undefined. Options or other settings could change the order of evaluation, so please do not code expressions with side effects.

(6) Structure and union specifiers
These are adjusted so that they do no span bit field type alignment boundaries. If packing has been conducting using options or a #pragma, then bit fields are packed, and not adjusted to alignment boundaries.

(7) Function definitions - storage of formal parameters
These are assigned to the stack and register. For details, refer to section 9.1.3 Rules Concerning Setting and Referencing Parameters.

(8) # operator
These are evaluated left to right.

4.1.2 Undefined Behavior

This section describes behavior that is not defined by the ANSI standard.

(1) Character set
A message is output if a source file contains a character not specified by the character set.

(2) Lexical elements
A message is output if there is a single or double quotation mark ("/") in the last category (a delimiter or a single non-whitespace character that does not lexically match another preprocessing lexical type).

(3) Identifiers
Since all identifier characters have meaning, there are no meaningless characters.

(4) Identifier binding
A message is output if both internal and external binding was performed on the same identifier within a translation unit.

(5) Compatible type and composite type
All declarations referencing the same object or function must be compatible. Otherwise, a message will be output.

(6) Character constants
Specific non-graphical characters can be expressed by means of extended notation, consisting of a backslash (\) followed by a lower-case letter. The following are available: \a, \b, \f, \n, \r, \t, and \v. There is no other extended notation; other letters following a backslash (\) become that letter.

(7) String literals - concatenation
When a simple string literal is adjacent to a wide string literal token, simple string concatenation is performed.

(8) String literals - modification
Users modify string literals at their own risk. Although the string will be changed if it is allocated to RAM, it will not be changed if it is allocated to ROM.

(9) Header names
If the following characters appear in strings between the delimiter characters < and >, or between two double quotation marks ("), then they are treated as part of the file name: characters, comma (,), double quote ("), two slashes (//), or slash-asterisk (/*). The backslash (\) is treated as a folder separator.

(10) Floating point type and integral type
If a floating-point type is converted into an integral type, and the integer portion cannot be expressed as an integral type, then the value is truncated until it can.

(11) lvalues and function specifiers
A message is output if an incomplete type becomes an lvalue.

(12) Function calls - number of arguments
If there are too few arguments, then the values of the formal parameters will be undefined. If there are too many arguments, then the excess arguments will be ignored when the function is executed, and will have no effect.
A message will be output if there is a function declaration before the function call.

(13) Function calls - types of extended parameters
If a function is defined without a function prototype, and the types of the extended arguments do not match the types of the extended formal parameters, then the values of the formal parameters will be undefined.

(14) Function calls - incompatible types
If a function is defined with a type that is not compatible with the type specified by the expression indicating the called function, then the return value of the function will be invalid.

(15) Function declarations - incompatible types
If a function is defined in a form that includes a function prototype, and the type of an extended argument is not compatible with that of a formal parameter, or if the function prototype ends with an ellipsis, then it will be interpreted as the type of the formal parameter.

(16) Addresses and indirection operators
If an incorrect value is assigned to a pointer, then the behavior of the unary * operator will either obtain an undefined value or result in an illegal access, depending on the hardware design and the contents of the incorrect value.

(17) Cast operator - function pointer casts
If a typecast pointer is used to call a function with other than the original type, then it is possible to call the function. If the parameters or return value are not compatible, then it will be invalid.

(18) Cast operator - integral type casts
If a pointer is cast into an integral type, and the amount of storage is too small, then the storage of the cast type will be truncated.

(19) Multiplicative operators
A message will be output if a divide by zero is detected during compilation.
During execution, a divide by zero will raise an exception. If an error-handling routine has been coded, it will be handled by this routine.

(20) Additive operators - non-array pointers
If addition or subtraction is performed on a pointer that does other than indicate elements in an array object, the behavior will be as if the pointer indicates an array element.

(21) Additive operators - subtracting a pointer from another array
If subtraction is performed using two pointers that do not indicate elements in the same array object, the behavior will be as if the pointers indicate array elements.

(22) Bitwise shift operators
If the value of the right operand is negative, or greater than the bit width of the extended left operand, then the result will be the shifted value of the right operand, masked by the bit width of the left operand.

(23) Relational operators - pointers
If the objects pointed to by the pointers being compared are not members of the same structure or union object, then the relational operation will be performed as if it were for pointers pointing to members of the same structure or union object.

(24) Simple assignment
If a value stored in an object is accessed via another object that overlaps that object's storage area in some way, then the overlapping portion must match exactly. Furthermore, the types of the two objects must have modified or non-modified versions with compatible types. Assignment to non-matching overlapping storage could cause the value of the assignment source to become corrupted.

(25) Structure and union specifiers
If the member declaration list does not include named members, then a message will be output warning that the list has no effect. Note, however, that the same message will be output accompanied by an error if the -Xansi option is specified.

(26) Type modifiers - const
A message will be output if an attempt is made to modify an object defined with a const modifier, using an lvalue that is the non-const modified version. Casting is also prohibited.

(27) Type modifiers - volatile
A message will be output if an attempt is made to modify an object defined with a volatile modifier, using an lvalue that is the non-volatile modified version.

(28) return statements
A message will be output if a return statement without an expression is executed, and the caller uses the return value of the function, and there is a declaration. If there is no declaration, then the return value of the function will be undefined.

(29) Function definitions
If a function taking a variable number of arguments is defined without a parameter type list that ends with an ellipsis, then the values of the formal parameters will be undefined.

(30) Conditional inclusion
If a replacement operation generates a "defined" token, or if the usage of the "defined" unary operator before macro replacement does not match one of the two formats specified in the constraints, then it will be handled as an ordinary "defined".

(31) Macro replacement - arguments not containing preprocessing tokens
A message is output if the arguments (before argument replacement) do not contain preprocessing tokens.

(32) Macro replacement - arguments with preprocessing directives
A message is output if an argument list contains a preprocessor token stream that would function as a processing directive in another circumstance.

(33) # operator
A message is output if the results of replacement are not a correct simple string literal.

(34) ## operator
A message is output if the results of replacement are not a correct simple string literal.

4.1.3 Implementation-defined behavior of C90

(1) How to identify diagnostic messages (5.1.1.3).
Refer to "10. MESSAGES".

(2) The semantics of the arguments to main (5.1.2.2.1).
Not defined.

(3) What constitutes an interactive device (5.1.2.3).
Not defined for the configuration of an interactive device.

(4) The number of significant initial characters (beyond 31) in an identifier without external linkage (6.1.2).
The first 8,189 characters are handled as significant characters.

(5) The number of significant initial characters (beyond 6) in an identifier with external linkage (6.1.2).

The first 8,191 characters are handled as significant characters.

(6) Whether case distinctions are significant in an identifier with external linkage (6.1.2).
Characters in an identifier are case sensitive.

(7) The members of the source and execution character sets, except as explicitly specified in the Standard (5.2.1).
The values of the members of the source and execution character sets are ASCII code, EUC, SJIS, UTF-8, big5, and gb2312. Japanese and Chinese descriptions in comments and strings are supported.

(8) The shift states used for the encoding of multibyte characters (5.2.1.2).
A representation format that depends on the shift status is not supported.

(9) The number of bits in a character in the execution character set (5.2.4.2.1).
A character consists of 8 bits. A multibyte character consists of 16 bits.

(10) The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.1.3.4).
The members of the source character set match the members of the execution character set.

(11) The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant (6.1.3.4).
Specific non-graphical characters can be represented by means of extended notation, consisting of a backslash (\) followed by a lower-case letter. The following are available: \a, \b, \f, \n, \r, \t, and \v. There is no other extended notation; other letters following a backslash (\) become that letter.

Escape sequence	Value (ASCII)
\a	0x07
\b	0x08
\f	0x0C
\n	0x0A
\r	0x0D
\t	0x09
\v	0x0B

(12) The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (6.1.3.4).
If an integer character constant contains four-byte characters (equivalent to four characters in ASCII), all the four bytes are valid values.
If the integer character constant contains five or more bytes, an error message is output.

(13) The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (6.1.3.4).
No locale is supported.

(14) Whether a "plain" char has the same range of values as signed char or unsigned char (6.2.1.1).
The char type has the same range of values, the same representation format, and the same behavior as those of the unsigned char type.
However, the char type can be switched to the signed char type by using the **-signed_char** option.

(15) The representations and sets of values of the various types of integers (6.1.2.5).
Refer to "4.1.5 Internal Data Representation and Areas".

(16) The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2).
If an integer is converted to an integer containing less bits, the resultant bit string is masked by the bit width of the less number of bits (with the upper bits deleted). If an unsigned integer is converted to a signed integer containing the same number of bits, the bit string is copied as is.

(17) The results of bitwise operations on signed integers (6.3).
Arithmetic shift is performed for a shift operator. For other operators, a signed integer is calculated as an unsigned value (as a bit image).

(18) The sign of the remainder on integer division (6.3.5).

If an operand has a negative value, the result of the "%" operator has the same sign as that of the first operand (dividend).

(19) The result of a right shift of a negative-valued signed integral type (6.3.7).
In the case of "E1 >> E2", if E1 is a negative-valued signed type, arithmetic shift is performed.

(20) The representations and sets of values of the various types of floating-point numbers (6.1.2.5).
Refer to "4.1.5 Internal Data Representation and Areas".

(21) The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (6.2.1.3).
Rounded in the direction of the nearest value if any of these conditions is met:
-nofpu is specified.
-cpu=rx200 is specified.
-cpu=rx600 or -isa=rxv1 is specified, and the integer to be converted is unsigned.
In other cases, the result of conversion to the single-precision floating-point type is in accordance with the setting of the RM[0:1] bit in the FPSW.
[V3.01.00 or later] If -dpfpu is specified, the result of conversion to the double-precision floating-point type is in accordance with the setting of the DRM[0:1] bit in the DPSW.

(22) The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (6.2.1.4).
Rounded to the nearest representable direction.

(23) The type of integer required to hold the maximum size of an array --- that is, the type of the sizeof operator, size_t (6.3.3.4, 7.1.1).
Unsigned long type

(24) The result of casting a pointer to an integer or vice versa (6.3.4).
- Integer-to-pointer conversion result
  If the size of an integer type is larger than or equal to that of a pointer type, the lower-byte value of the integer type is used. If the size of the integer type is smaller than that of the pointer type, the sign-extended value is used.
- Pointer-to-integer conversion result
  If the size of a pointer type is larger than or equal to that of an integer type, the lower-byte value of the pointer type is used. If the size of a pointer type is smaller than that of an integer type, the zero-extended value is used.

(25) The type of integer required to hold the difference between two pointers to members of the same array, ptrdiff_t (6.3.4, 7.1.1).
long type

(26) The extent to which objects can actually be placed in registers by use of the register storage-class specifier (6.5.1).
Optimization is performed so that objects are accessed as fast as possible regardless of whether the register storage-class specifier is declared.

(27) A member of a union object is accessed using a member of a different type (6.3.2.3).
The internal data representation is determined by the type that is to be accessed.

(28) The padding and alignment of members of structures (6.5.2.1).
Refer to "4.1.5 Internal Data Representation and Areas".

(29) Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field (6.5.2.1).
A bit field that was declared without specifying a sign is treated as an unsigned int type.

(30) The order of allocation of bit-fields within an int (6.5.2.1).
The first declared bit-field is allocated from the lowest-order bit in the area with the size of the type when the bit-field was declared. This can be changed by the option.
Refer to "4.1.5 Internal Data Representation and Areas".

(31) Whether a bit-field can straddle a storage-unit boundary (6.5.2.1).
A bit-field is allocated to the next area instead of straddling a storage-unit boundary.
Refer to "4.1.5 Internal Data Representation and Areas".

(32) The integer type chosen to represent the values of an enumeration type (6.5.2.2).
Signed long type. If the **-auto_enum** option is specified, the minimum type that an enumerated type fits in is used.

(33) What constitutes an access to an object that has volatile-qualified type (6.5.3).
The order and the number of accesses are as described in the C source. However, this does not apply to access to a type for which the microcomputer does not have a corresponding instruction. Such types might be accessed in a smaller size than a declaration type.

(34) The maximum number of declarators that may modify an arithmetic, structure, or union type (6.5.4).
128

(35) The maximum number of case values in a switch statement (6.6.4.2).
There are no limits.

(36) Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).
The value for the character constant specified in conditional inclusion matches the character constant value that appears in other expressions.
Such a character constant does not have a negative value.

(37) The method for locating includable source files (6.8.2).
Files are searched in the following order, and a file having the same name in the folder is identified as the header:
1. Folder specified by the path (if it is full-path). If full-path is not specified:
2. Folder having the source file
3. Folder specified by the **-include** option
4. Standard include file folder specified in the INC_RXA environment variable

(38) The support for quoted names for includable source files (6.8.2).
Files are searched in the following order:
1. Folder specified by the path (if it is full-path). If full-path is not specified:
2. Folder having the source file
3. Folder specified by the **-include** option
4. Standard include file folder specified in the INC_RXA environment variable

(39) The mapping of source file character sequences (6.8.2).
A preprocessing token string enclosed in < and > or in double quotation marks (") is mapped to the header name as is. If the preprocessing token string is converted to the <character string> or "character string" format by expanding a macro, the content is mapped to the header name.

(40) The behavior on each recognized #pragma directive (6.8.6).
Refer to "4.2.3 #pragma Directive".

(41) The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.8.8).
A date and time are always obtained.

(42) The null pointer constant to which the macro NULL expands (7.1.6).
0

(43) The diagnostic printed by and the termination behavior of the assert function (7.2).
The display varies depending on the **-lang** option specified at compilation.
If -lang=c99 is not specified (for C(C89), C++, or EC++ language):
ASSERTION FAILED: *Expression* FILE *file-name*,LINE *line-number*
If -lang=c99 is specified (for C(C99) language):
ASSERTION FAILED: *Expression* FILE *file-name*,LINE *line-number* FUNCNAME *function-name*
The behavior when the assert function ends is not defined. As per low-level interface routine specifications.

(44) The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, and isupper functions (7.3.1).
Unsigned char type (0 to 255) and EOF (-1)

(45) The values returned by the mathematics functions on domain errors (7.5.1).
A NaN is returned.
Refer to "(5) Floating-Point Number Specifications" in "4.1.5 Internal Data Representation and Areas".

(46) Whether the mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors (7.5.1).
For details about the functions that set ERANGE in errno when an underflow occurs, refer to "10.5.6 Standard Library Error Messages". Other functions do not set ERANGE in errno.

(47) Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero (7.5.6.4).
A domain error occurs. For details, see the description about the fmod function group.

(48) The set of signals for the signal function (7.7.1.1).
The signal function is not supported.

(49) The semantics for each signal recognized by the signal function (7.7.1.1).
The signal function is not supported.

(50) The default handling and the handling at program startup for each signal recognized by the signal function (7.7.1.1).
The signal function is not supported.

(51) If the equivalent of signal(sig, SIG_DFL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed (7.7.1.1).
The signal function is not supported.

(52) Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function (7.7.1.1).
The signal function is not supported.

(53) Whether the last line of a text stream requires a terminating new-line character (7.9.2).
Not defined. As per low-level interface routine specifications.

(54) Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.9.2).
Not defined. As per low-level interface routine specifications.

(55) The number of null characters that may be appended to data written to a binary stream (7.9.2).
Not defined. As per low-level interface routine specifications.

(56) Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file (7.9.3).
Not defined. As per low-level interface routine specifications.

(57) Whether a write on a text stream causes the associated file to be truncated beyond that point (7.9.3).
Not defined. As per low-level interface routine specifications.

(58) The characteristics of file buffering (7.9.3).
Not defined. As per low-level interface routine specifications.

(59) Whether a zero-length file actually exists (7.9.3).
Not defined. As per low-level interface routine specifications.

(60) The rules for composing valid file names (7.9.3).
Not defined. As per low-level interface routine specifications.

(61) Whether the same file can be open multiple times (7.9.3).
Not defined. As per low-level interface routine specifications.

(62) The effect of the remove function on an open file (7.9.4.1).
The remove function is not supported.

(63) The effect if a file with the new name exists prior to a call to the rename function (7.9.4.2).
The rename function is not supported.

(64) The output for %p conversion in the fprintf function (7.9.6.1).
Hexadecimal output

(65) The input for %p conversion in the fscanf function (7.9.6.2).
Hexadecimal input

(66) The interpretation of a - character that is neither the first nor the last character in the scan list for %[conversion in the fscanf function (7.9.6.2).
The - character indicates the range between the previous character and the next character unless the previous character is a ^ character.

(67) The value to which the macro errno is set by the fgetpos or ftell function on failure (7.9.9.1, 7.9.9.4).
The fgetpos function is not supported.
Not defined for the ftell function. As per low-level interface routine specifications.

(68) The messages generated by the perror function (7.9.10.4).
Refer to "7.4 Library Function".

(69) The behavior of the calloc, malloc, or realloc function if the size requested is zero (7.10.3).
NULL is returned.

(70) The behavior of the abort function with regard to open and temporary files (7.10.4.1).
Not defined. As per low-level interface routine specifications.

(71) The status returned by the exit function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE (7.10.4.3).
The exit function is not supported.

(72) The set of environment names and the method for altering the environment list used by the getenv function (7.10.4.4).
The getenv function is not supported.

(73) The contents and mode of execution of the string by the system function (7.10.4.5).
The system function is not supported.

(74) The contents of the error message strings returned by the strerror function (7.11.6.2).
Refer to "10.5.6 Standard Library Error Messages".

(75) The local time zone and Daylight Saving Time (7.12.1).
The time.h is not supported.

(76) The era for the clock function (7.12.2.1).
The clock function is not supported.

4.1.4 Implementation-defined behavior of C99

(1) How a diagnostic is identified (3.10, 5.1.1.3).
Refer to "10. MESSAGES".

(2) Whether each non-empty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).
Retained as they are.

(3) The mapping between physical source file multi-byte characters and the source character set in translation phase 1 (5.1.1.2).
Multibyte characters are mapped to the corresponding source character set according to the compile option.

(4) The name and type of the function called at program startup in a freestanding environment (5.1.2.1).
Not defined. Depends on the startup implementation.

(5) The effect of program termination in a freestanding environment (5.1.2.1).
Depends on startup in a normal termination. Create an exit processing routine if the program needs to terminate abnormally.

(6) An alternative manner in which the main function may be defined (5.1.2.2.1).
Not defined because of a freestanding environment.

(7) The values given to the strings pointed to by the argv argument to main (5.1.2.2.1).
Not defined because of a freestanding environment.

(8) What constitutes an interactive device (5.1.2.3).
Not defined for the configuration of an interactive device.

(9) The set of signals, their semantics, and their default handling (7.14).
The signal handling functions are not supported.

(10) Signal values other than SIGFPE, SIGILL, and SIGSEGV that correspond to a computational exception (7.14.1.1).
The signal handling functions are not supported.

(11) Signals for which the equivalent of signal(sig, SIG_IGN); is executed at program startup (7.14.1.1).
The signal handling functions are not supported.

(12) The set of environment names and the method for altering the environment list used by the getenv function (7.20.4.5).
The getenv function is not supported.

(13) The manner of execution of the string by the system function (7.20.4.6).
The system function is not supported.

(14) Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).
Multibyte characters cannot be used as identifiers.

(15) The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).
The entire identifier is handled as meaningful. The length of an identifier is unlimited.

(16) The number of bits in a byte (3.6).
8 bits.

(17) The values of the members of the execution character set (5.2.1).

The element values of the execution character set are ASCII code, EUC, SJIS, UTF-8, big5 and gb2312 values.

(18) The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

Escape Sequence	Value (ASCII)
"\a"	0x07
"\b"	0x08
"\f"	0x0C
"\n"	0x0A
"\r"	0x0D
"\t"	0x09
"\v"	0x0B

(19) The value of a char object into which has been stored any character other than a member of the basic execution character set (6.2.5).
Value that is type-converted to char type.

(20) Which of signed char or unsigned char has the same range, representation, and behavior as "plain" char (6.2.5, 6.3.1.1).
The char type has the same range of values, the same representation format and the same behavior as the unsigned char type. However, it can be switched to the signed char type by the **-signed_char** option.

(21) The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).
When the character code selection of the input program is the same as the character code selection of the output file according to an option specification, association with the element having the same value is performed. If they are different according to the option specification, the value of the corresponding character code is used.

(22) The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).
A integer character constant consisting of up to four characters has a four-byte value with the lower byte being the last character and the upper byte being the first character. A character constant having five or more characters results in an error. A character which is not represented by basic execution environment character set is regarded as a integer character constant having that value. In an invalid escape sequence, the backslash is ignored and the next character is regarded as a integer character constant.

(23) The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).
Left-most character value as a multibyte character.

(24) The current locale used to convert a wide character constant consisting of a single multi-byte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).
Locale is not supported.

(25) The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).
Locale is not supported.

(26) The value of a string literal containing a multi-byte character or escape sequence not represented in the execution character set (6.4.5).
Corresponding byte value for escape sequence or corresponding each byte value for a multibyte character.

(27) Any extended integer types that exist in the implementation (6.2.5).
No extended integer types are provided.

(28) Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).
The signed integer type is represented in two's complement, and there are no trap representations.

(29) The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).
No extended integer types are provided.

(30) The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

Bit string masked by the width of the conversion target type (with the upper bits truncated).

(31) The results of some bit-wise operations on signed integers (6.5).
Arithmetic shift is performed for a shift operator. For other operators, a signed integer is calculated as an unsigned value (as a bit image).

(32) The accuracy of the floating-point operations and of the library functions in <math.h> and <complex.h> that return floating-point results (5.2.4.2.2).
Unknown.

(33) The rounding behaviors characterized by non-standard values of FLT_ROUNDS (5.2.4.2.2).
No nonstandard value is defined for FLT_ROUNDS.

(34) The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD (5.2.4.2.2).
No nonstandard value is defined for FLT_EVAL_METHOD.

(35) The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).
Rounded to the nearest direction if any of these conditions is met:
-nofpu is specified.
-cpu=rx200 is specified.
-cpu=rx600 or -isa=rxv1 is specified and the integer to be converted is unsigned.
Otherwise, the setting in the RM[0:1] bits in FPSW is followed.
[V3.01.00 or later] When -dpfpu is specified, the result of conversion to the double-precision floating-point type is in accord with the setting of the DRM[0:1] bits in the DPSW.

(36) The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).
As per the option (**-round**) specification and microcomputer settings.

(37) How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).
As per the option (**-round**) specification.

(38) Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma (6.5).
Contraction of expressions depends on each option specification.
The FP_CONTRACT pragma does not work.
#pragma STDC FP_CONTRACT is ignored even if it is specified.

(39) The default state for the FENV_ACCESS pragma (7.6.1).
The default state of the FENV_ACCESS pragma is ON.
#pragma STDC FENV_ACCESS is ignored even if it is specified.

(40) Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).
As per the math.h and fenv.h libraries provided by the compiler. There are no additional definitions.

(41) The default state for the FP_CONTRACT pragma (7.12.2).
The default state of the FP_CONTRACT pragma is ON.

(42) Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).
No "inexact" floating-point exception is generated for the nearbyint, nearbyintf or nearbyintl function. An "inexact" floating-point exception may be generated for the rint, rintf or rintl function.

(43) Whether the underflow (and inexact) floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9).
As per the option (**-round**) specification and microcomputer settings.

(44) The result of converting a pointer to an integer or vice versa (6.3.2.3).
- Integer-to-pointer conversion result
If the size of an integer type is larger than that of a pointer type, the lower-byte value of the integer type is used. If the size of the integer type is equal to that of the pointer type, the bit pattern of the integer type is retained as is. If the size of the integer type is smaller than that of the pointer type, the resultant value of an extension to a long type is retained as is.
- Pointer-to-integer conversion result
If the size of a pointer type is larger than that of an integer type, the lower-byte value of the pointer type is used. If the size of the pointer type is equal to that of the integer type, the bit pattern of the pointer type is retained as is. If the size of a pointer type is smaller than that of an integer type, the zero-extended value of the pointer type is used.

(45) The size of the result of subtracting two pointers to elements of the same array (6.5.6).

The resultant type is the signed long type.

(46) The extent to which suggestions made by using the register storage-class specifier are effective (6.7.1).
User requests for register variables are not honored.

(47) The extent to which suggestions made by using the inline function specifier are effective (6.7.4).
Expansion is attempted only when the -inline option is valid. In addition, expansion may not be performed depending on the condition.

(48) Whether a "plain" int bit-field is treated as signed int bit-field or as an unsigned int bit-field (6.7.2, 6.7.2.1).
Treated as an unsigned int type. However, this can be changed by the **-signed_bitfield** option.

(49) Allowable bit-field types other than _Bool, signed int, and unsigned int (6.7.2.1).
All integer types are allowed.

(50) Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).
When structure type packing is not specified, a bit-field cannot straddle a strage-unit boundary, but it is allocated to the next area.
When structure type packing is specified, a bit-field may straddle a strage-unit boundary.

(51) The order of allocation of bit-fields within a unit (6.7.2.1).
Allocated from the lower order. Selectable by the **-bit_order** option or #pragma bit_order.

(52) The alignment of non-bit-field members of structures (6.7.2.1).
Refer to "4.1.5 Internal Data Representation and Areas".

(53) The integer type compatible with each enumerated type (6.7.2.2).
Signed long type. However, the minimum type that an enumerated type fits in if the **-auto_enum** option is specified.

(54) What constitutes an access to an object that has volatile-qualified type (6.7.3).
Although the order and number of accesses are as described in the C source, this does not apply to those accesses to a type for which the microcomputer does not have a corresponding instruction. An object qualified as volatile might be accessed in a smaller size than a declaration type.

(55) How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).
A character string described in the #include is interpreted as the character code specified as the source character set and is associated with a header name or an external source file name.

(56) Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).
A value for the character constant specified in conditional inclusion is equal to the character constant value that appears in other expressions.

(57) Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

(58) The places that are searched for an included < > delimited header, and how the places are specified other header is identified (6.10.2).
Folders are searched in this order and a file having the same name in the folder is identified as the header.
1. Folder specified by the path (if it is full-path)
2. Folder specified by the **-include** option
3. Standard include file folder (..\inc folder with a relative path from the bin folder where the compiler is placed)

(59) How the named source file is searched for in an included " " delimited header (6.10.2).
Searched in this order:
1. Folder specified by the path (if it is full-path)
2. Folder having the source file
3. Folder specified by the **-include** option
4. Standard include file folder (..\inc folder with a relative path from the bin folder where the compiler is placed)

(60) The method by which preprocessing tokens (possibly resulting from macro expansion) in a #include directive are combined into a header name (6.10.2).
Treated as a preprocessing token of a single header or file name only in a macro that replaces preprocessing tokens with a single <character string> or "character string" format.

(61) The nesting limit for #include processing (6.10.2).
There are no limits.

(62) Whether the # operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.3.2).
A \ character is not inserted in front of the first \ character.

(63) The behavior on each recognized non-STDC #pragma directive (6.10.6).
Refer to "4.2.3 #pragma Directive".

(64) The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.10.8).
A date and time are always obtained.

(65) Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).
Refer to "7. LIBRARY FUNCTIONAL SPECIFICATION".

(66) The format of the diagnostic printed by the assert macro (7.2.1.1).
As follows:
Assertion failed: *Expression* FILE *file-name,*LINE *line-number* FUNCNAME *function-name*

(67) The representation of the floating-point status flags stored by the fegetexceptflag function (7.6.2.2).
One or combination of the following:

_FE_DIVBYZERO	0x04
_FE_INEXACT	0x10
_FE_INVALID	0x01
_FE_OVERFLOW	0x02
_FE_UNDERFLOW	0x08

(68) Whether the feraiseexcept function raises the "inexact" floating-point exception in addition to the "overflow" or "underflow" floating-point exception (7.6.2.3).
An "inexact" floating-point exception may be generated if the E0 flag is set to 0 when an "overflow" floating-point exception is generated. Otherwise, no "inexact" floating-point exception is generated.

(69) Strings other than "C" and "" that may be passed as the second argument to the setlocale function (7.11.1.1).
The setlocale function is not supported.

(70) The types defined for float_t and double_t when the value of the FLT_EVAL_METHOD macro is less than zero or greater than two (7.12).
float_t is defined as the float type and double_t as the double type.

(71) Domain errors for the mathematics functions, other that those required by this International Standard (7.12.1).
Refer to (72).

(72) The values returned by the mathematics functions on domain errors (7.12.1).

acosh function group	The argument is x. If x < 1 is true, a domain error occurs. In this case, the return value is a NaN.
atanh function group	The argument is x. If -1 < x < 1 is true, a domain error occurs. In this case, the return value is a NaN.
ccosh function group	The argument is x. - If the real part of x is 0 and the imaginary part of x is ±∞, a domain error occurs. In this case, the return value is NaN + 0 × i. - If the real part of x is ±∞ and the imaginary part of x is ±∞, a domain error occurs. In this case, the return value is ∞ + NaN × i. - If the real part of x is a finite value and the imaginary part of x is ±∞, a domain error occurs. In this case, the return value is NaN + NaN × i.
cexp function group	The argument is x. - If the real part of x is +∞ and the imaginary part of x is ±∞, a domain error occurs. In this case, the return value is +∞ + NaN × i. - If the real part of x is a finite value and the imaginary part of x is ±∞, a domain error occurs. In this case, the return value is NaN + NaN × i
csinh function group	The argument is x. - If the real part of x is 0 or ±∞ and the imaginary part of x is ±∞, a domain error occurs. In this case, the return value is the real part of x + NaN × i. - If the real part of x is a finite value and the imaginary part of x is ±∞, a domain error occurs. In this case, the return value is NaN + NaN × i.

fma function group	The first argument is x1, the second argument is x2, and the third argument is x3. - If x1 is ±∞, x2 is 0, and x3 is not a NaN, a domain error occurs. In this case, the return value is a NaN. - If x1 is 0 , x2 is ±∞, and x3 is not a NaN, a domain error occurs (no definition). In this case, the return value is a NaN. - If x1, x2, and x3 are ±∞ and the sign of x1 and x2 is different from that of x3, (that is, subtraction between ∞ occurs), a domain error occurs. In this case, the return value is a NaN.
llrint function group	The argument is x. - If x is a finite value that cannot be represented in the longlong type, a domain error occurs. In this case, the return value is 0. - If x is a NaN or ±∞, a domain error occurs. In this case, the return value is 0.
llround function group	The argument is x. - If x is a finite value that cannot be represented in the longlong type, a domain error occurs. In this case, the return value is 0. - If x is a NaN or ±∞, a domain error occurs. In this case, the return value is 0.
log1p function group	The argument is x. If x satisfies x < -1, a domain error occurs. In this case, the return value is a NaN.
lrint function group	The argument is x. - If x is a finite value that cannot be represented in the long type, a domain error occurs. In this case, the return value is 0. - If x is a NaN or ±∞, a domain error occurs. In this case, the return value is 0.
lround function group	The argument is x. - If x is a finite value that cannot be represented in the long type, a domain error occurs. In this case, the return value is 0. - If x is a NaN or ±∞, a domain error occurs. In this case, the return value is 0.
remquo function group	The first argument is x1, the second argument is x2, and the third argument is x3. - If x1 is ±∞ and x2 is not a NaN, a domain error occurs. In this case, the return value is a NaN. Note that x3 points to 0. - If x1 is not a NaN and x2 is 0, a domain error occurs. In this case, the return value is a NaN. Note that x3 points to 0.
tgamma function group	The argument is x. If x is -∞ or a negative integer, a domain error occurs. In this case, the return value is a NaN.
carg function group	The argument is x. If the real part and imaginary part of x are 0, a domain error occurs. In this case, the return value is a NaN.
ccos function group	The argument is x. - If the real part of x is ±∞ and the imaginary part of x is 0, a domain error occurs. In this case, the return value is NaN + 0 × i. - If the real part of x is ±∞ and the imaginary part of x is ±∞, a domain error occurs. In this case, the return value is ∞ + NaN × i. - If the real part of x is ±∞ and the imaginary part of x is a finite value, a domain error occurs. In this case, the return value is NaN + NaN × i.
clog function group	The argument is x. If the real part of x is ±∞ and the imaginary part of x is ±∞, a domain error occurs. In this case, the return value is ±∞ + NaN × i.

cpow function group	The first argument is x1 and the second argument is x2. - If the imaginary part of x1 is 0 and, as a result of multiplying x2 by the real part of x1, the real part is +∞ and the imaginary part is ±∞, a domain error occurs. In this case, the return value is +∞ + NaN × i. - If the imaginary part of x1 is 0 and, as a result of multiplying x2 by the real part of x1, the real part is a finite value and the imaginary part is ±∞, a domain error occurs. In this case, the return value is NaN + NaN × i. - If the real part of x1 or x2 is a NaN and the imaginary parts of x1 and x2 are 0, a domain error occurs. In this case, the return value is NaN + 0 × i. - If the real part of x1 is 0, the real part of x2 is 0 or smaller, and the imaginary parts of x1 and x2 are 0, a domain error occurs. In this case, the return value is NaN + 0 × i. - If the real part of x1 is a negative number, the real part of x2 is a noninteger, and the imaginary parts of x1 and x2 are 0, a domain error occurs. In this case, the return value is NaN + 0 × i.
remainder function group	The first argument is x1 and the second argument is x2. - If x1 is ±∞ and x2 is not a NaN, a domain error occurs. In this case, the return value is a NaN. - If x1 is not a NaN and x2 is 0, a domain error occurs. In this case, the return value is a NaN.
acos function group	The argument is x. - If x satisfies x < -1 or 1 < x, a domain error occurs. In this case, the return value is a NaN. - If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
asin function group	The argument is x. - If x satisfies x < -1 or 1 < x, a domain error occurs. In this case, the return value is a NaN. - If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
atan function group	The argument is x. If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
cosh function group	The argument is x. If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
exp function group	The argument is x. If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
frexp function group	The first argument is x1 and the second argument is x2. If x1 is a NaN, a domain error occurs. In this case, the return value is a NaN. Note that the pointing target of x2 does not change.
log10 function group	The argument is x. - If x < 0 is true, a domain error occurs. In this case, the return value is a NaN. - If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
log function group	The argument is x. - If x is a NaN, a domain error occurs. In this case, the return value is a NaN. - If x < 0 is true, a domain error occurs. In this case, the return value is a NaN.
sin function group	The argument is x. If x is ±∞ or NaN, a domain error occurs. In this case, the return value is a NaN.
cos function group	The argument is x. If x is ±∞ or NaN, a domain error occurs. In this case, the return value is a NaN.
sinh function group	The argument is x. If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
tan function group	The argument is x. If x is ±∞ or NaN, a domain error occurs. In this case, the return value is a NaN.

tanh function group	The argument is x. If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
atan2 function group	The first argument is x1 and the second argument is x2. - If x1 and x2 are 0, a domain error occurs. In this case, the return value is a NaN. - If x1 or x2 is a NaN, a domain error occurs. In this case, the return value is a NaN. - If x1 and X2 are ±∞, a domain error occurs. In this case, the return value is a NaN.
ceil function group	The argument is x. If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
floor function group	The argument is x. If x is a NaN, a domain error occurs. In this case, the return value is a NaN.
fmod function group	The first argument is x1 and the second argument is x2. - If x2 is 0, a domain error occurs. In this case, the return value is a NaN. - If x1 or x2 is a NaN, a domain error occurs. In this case, the return value is a NaN. - If x1 is ±∞, a domain error occurs. In this case, the return value is a NaN.
ldexp function group	The first argument is x1 and the second argument is x2. If x1 is a NaN, a domain error occurs. In this case, the return value is a NaN.
modf function group	The first argument is x1 and the second argument is x2. If x1 is a NaN, a domain error occurs. In this case, the return value is a NaN. Note that x2 points to the NaN.
pow function group	The first argument is x1 and the second argument is x2. - If x1 or x2 is a NaN, a domain error occurs. In this case, the return value is a NaN. - If x1 is 0 and x2 is 0 or smaller, a domain error occurs. In this case, the return value is a NaN. - If x1 satisfies x1 < 0 and x2 is not an integer, a domain error occurs. In this case, the return value is a NaN.
sqrt function group	The argument is x. - If x satisfies x < 0, a domain error occurs. In this case, the return value is a NaN. - If x is a NaN, a domain error occurs. In this case, the return value is a NaN.

(73) The values returned by the mathematics functions on underflow range errors, whether errno is set to the value of the macro ERANGE when the integer expression math_errhandling & MATH_ERRNO is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression math_errhandling & MATH_ERREXCEPT is nonzero. (7.12.1).
The return value is 0. ERANGE is set in errno in case of an underflow. An "underflow" floating-point exception is generated.

(74) Whether a domain error occurs or zero is returned when an fmod function has a second argument of zero (7.12.10.1).
A domain error is generated. For details, see the description about the fmod function group.

(75) The base-2 logarithm of the modulus used by the remquo functions in reducing the quotient (7.12.10.3).
31.

(76) Whether the equivalent of signal(sig, SIG_DFL); is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).
The signal handling functions are not supported.

(77) The null pointer constant to which the macro NULL expands (7.17).
0.

(78) Whether the last line of a text stream requires a terminating new-line character (7.19.2).
As per low-level interface routine specifications.

(79) Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).
As per low-level interface routine specifications.

(80) The number of null characters that may be appended to data written to a binary stream (7.19.2).
As per low-level interface routine specifications.

(81) Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3).
As per low-level interface routine specifications.

(82) Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).
As per low-level interface routine specifications.

(83) The characteristics of file buffering (7.19.3).
As per low-level interface routine specifications.

(84) Whether a zero-length file actually exists (7.19.3).
As per low-level interface routine specifications.

(85) The rules for composing valid file names (7.19.3).
As per low-level interface routine specifications.

(86) Whether the same file can be simultaneously open multiple times (7.19.3).
As per low-level interface routine specifications.

(87) The nature and choice of encodings used for multibyte characters in files (7.19.3).
The shift state is not supported as the representation form of multibyte characters.

(88) The effect of the remove function on an open file (7.19.4.1).
The remove function is not supported.

(89) The effect if a file with the new name exists prior to a call to the rename function (7.19.4.2).
The rename function is not supported.

(90) Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).
The tmpfile function is not supported.

(91) Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).
If filename is a null pointer, it changes the mode of the current stream to the specified mode.

(92) The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.19.6.1, 7.24.2.1).
++++++ is output for the positive infinity, ------ for a negative infinity, and ****** for a NaN.
n character strings or n wide character strings are not supported when a NaN is written.

(93) The output for %p conversion in the fprintf or fwprintf function (7.19.6.1, 7.24.2.1).
Hexadecimal notation.

(94) The interpretation of a - character that is neither the first nor the last character, nor the second where a ^ character is the first, in the scanlist for %[conversion in the fscanf() or fwscanf() function (7.19.6.2, 7.24.2.1).
Indicates a range between the previous character and the next character.

(95) The set of sequences matched by a %p conversion and the interpretation of the corresponding input item in the fscanf() or fwscanf() function (7.19.6.2, 7.24.2.2).
Hexadecimal number.

(96) The value to which the macro errno is set by the fgetpos, fsetpos, or ftell functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4).
The ftell function conforms with the low-level interface routine specifications.
The fgetpos and fsetpos functions are not supported.

(97) The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the strtod(), strtof(), strtold(), wcstod(), wcstof(), or wcstold() function (7.20.1.3, 7.24.4.1.1).
Interpreted as qNaN in case of the strtof, strtold, wcstod, wcstof, or wcstold function. Interpreted as a value other than a number of floating-point type in case of the strtod function.

(98) Whether or not the strtod, strtof, strtold, wcstod, wcstof, or wcstold function sets errno to ERANGE when underflow occurs (7.20.1.3, 7.24.4.1.1).
The strtod, strtof, strtold, wcstod, wcstof and wcstold functions set ERANGE in global variable errno.

(99) Whether the calloc, malloc, and realloc functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.20.3).

NULL is returned.

(100) Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the abort or _Exit function is called (7.20.4.1, 7.20.4.4).
As per low-level interface routine specifications.

(101) The termination status returned to the host environment by the abort, exit, or _Exit function (7.20.4.1, 7.20.4.3, 7.20.4.4).
The abort, exit and _Exit functions are not supported.

(102) The value returned by the system function when its argument is not a null pointer (7.20.4.6).
The system function is not supported.

(103) The local time zone and Daylight Saving Time (7.23.1).
time.h is not supported.

(104) The range and precision of times representable in clock_t and time_t (7.23).
time.h is not supported.

(105) The era for the clock function (7.23.2.1).
time.h is not supported.

(106) The replacement string for the %Z specifier to the strftime, and wcsftime functions in the "C" locale (7.23.3.5, 7.24.5.1).
time.h is not supported.

(107) Whether or when the trigonometric, hyperbolic, base-e exponential, base-e logarithmic, error, and log gamma functions raise the "inexact" floating-point exception in an IEC 60559 conformant implementation (F.9).
An "inexact" floating-point exception may be generated if the hypot, ldexp, lgamma, tgamma, erfc, pow, scalbln, tan, exp, or nexttoward function group resulted in an overflow or underflow.

(108) Whether the functions in <math.h> honor the rounding direction mode in an IEC 60559 conformant implementation (F.9).
The lround function group may not follow the rounding direction mode.

(109) The values or expressions assigned to the macros specified in the headers <float.h>, <limits.h>, and <stdint.h> (5.2.4.2, 7.18.2, 7.18.3).
Refer to each header and file in "7. LIBRARY FUNCTIONAL SPECIFICATION".

(110) The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).
Refer to "4.1.5 Internal Data Representation and Areas".

(111) The value of the result of the sizeof operator (6.5.3.4).
Refer to "4.1.5 Internal Data Representation and Areas".

Translation limits
The table below shows the translation limits of CC-RX.
The upper limit depends on the memory situation of the host environment for the item "No limit".

Item	Limit
Number of nesting levels of blocks	No limit
Number of nesting levels of conditional inclusion	No limit
Number of pointers, arrays, and function declarators (in any combinations) qualifying an arithmetic, structure, union, or incomplete type in a declaration	128
Number of nesting levels of parenthesized declarators within a full declarator	No limit
Number of nesting levels of parenthesized expressions within a full expression	No limit
Number of significant initial characters in an internal identifier or a macro name	No limit
Number of significant initial characters in an external identifier	No limit
Number of external identifiers in one translation unit	No limit
Number of identifiers with block scope declared in one block	No limit
Number of macro identifiers simultaneously defined in one preprocessing translation unit	No limit

Item	Limit
Number of parameters in one function definition	No limit
Number of arguments in one function call	No limit
Number of parameters in one macro definition	No limit
Number of arguments in one macro invocation	No limit
Number of characters in a logical source line	No limit
Number of characters in a character string literal or wide string literal (after concatenation)	No limit
Number of bytes in an object (in a hosted environment only)	2147483647
Number of nesting levels for #included files	No limit
Number of case labels for a switch statement (excluding those for any nested switch statements)	2147483647
Number of members in a single structure or union	No limit
Number of enumeration constants in a single enumeration	No limit
Number of levels of nested structure or union definitions in a single struct-declaration-list	No limit

4.1.5 Internal Data Representation and Areas

This section explains the data type and the internal data representation. The internal data representation is determined according to the following four items:

- Size
  Shows the memory size necessary to store the data.
- Boundary alignment
  Restricts the addresses to which data is allocated. There are three types of alignment; 1-byte alignment in which data can be allocated to any address, 2-byte alignment in which data is allocated to even byte addresses, and 4-byte alignment in which data is allocated to addresses of multiples of four bytes.
- Data range
  Shows the range of data of scalar type (C) or basic type (C++).
- Data allocation example
  Shows an example of assignment of element data of compound type (C) or class type (C++).

(1) Scalar Type (C), Basic Type (C++)
Table 4.1 shows internal representation of scalar type data in C and basic type data in C++.

Table 4.1 Internal Representation of Scalar-Type and Basic-Type Data

No	Data Type	Size (bytes)	Align-ment (bytes)	Signed/ Unsigned	Data Range	
					Minimum Value	Maximum Value
1	char *1	1	1	Unsigned	0	2^8–1 (255)
2	signed char	1	1	Signed	-2^7 (–128)	2^7–1 (127)
3	unsigned char	1	1	Unused	0	2^8–1 (255)
4	short	2	2	Signed	-2^{15} (–32768)	2^{15}–1 (32767)
5	signed short	2	2	Signed	-2^{15} (–32768)	2^{15}–1 (32767)
6	unsigned short	2	2	Unsigned	0	2^{16}–1 (65535)
7	int *2	4	4	Signed	-2^{31} (–2147483648)	2^{31}–1 (2147483647)

No	Data Type	Size (bytes)	Align-ment (bytes)	Signed/ Unsigned	Data Range	
					Minimum Value	Maximum Value
8	signed int *2	4	4	Signed	-2^{31} (–2147483648)	$2^{31}-1$ (2147483647)
9	unsigned int*2	4	4	Unsigned	0	$2^{32}-1$ (4294967295)
10	long	4	4	Signed	-2^{31} (–2147483648)	$2^{31}-1$ (2147483647)
11	signed long	4	4	Signed	-2^{31} (–2147483648)	$2^{31}-1$ (2147483647)
12	unsigned long	4	4	Unsigned	0	$2^{32}-1$ (4294967295)
13	long long	8	4	Signed	-2^{63} (–9223372036854775808)	$2^{63}-1$ (9223372036854775807)
14	signed long, long	8	4	Signed	-2^{63} (–9223372036854775808)	$2^{63}-1$ (9223372036854775807)
15	unsigned long, long	8	4	Unsigned	0	$2^{64}-1$ (18446744073709551615)
16	float	4	4	Signed	$-\infty$	$+\infty$
17	double, long double	4 *4	4	Signed	$-\infty$	$+\infty$
18	size_t	4	4	Unsigned	0	$2^{32}-1$ (4294967295)
19	ptrdiff_t	4	4	Signed	-2^{31} (–2147483648)	$2^{31}-1$ (2147483647)
20	enum*3	4	4	Signed	-2^{31} (–2147483648)	$2^{31}-1$ (2147483647)
21	Pointer	4	4	Unsigned	0	$2^{32}-1$ (4294967295)
22	bool *5 _Bool *8	1 *9	1 *9	-*9	-	-
23	Reference *6	4	4	Unsigned	0	$2^{32}-1$ (4294967295)
24	Pointer to a data member *6	4	4	Unsigned	0	$2^{32}-1$ (4294967295)
25	Pointer to a function member *6*7	12	4	- *10	-	-

Notes 1. When the **signed_char** option is specified, the **char** type has the same value range as the **signed char** type.

Notes 2. When the **int_to_short** option is specified, the **int** type has the same value range as the **short** type, the **signed int** type has the same value ranges as the **signed short** type, and the **unsigned int** type has the same value range as the **unsigned short** type.

Notes 3. When the **auto_enum** option is specified, the smallest type that holds enumeration values is selected.

Notes 4. When **dbl_size=8** is specified, the size of the **double** type and **long double** type is 8 bytes.

Notes 5. This data type is only valid for compilation of C++ programs or C programs including **stdbool.h.**

Notes 6. These data types are only valid for compilation of C++ programs.

Notes 7. Pointers to function and virtual function members are represented in the following data structure.

Notes 8. This data type is only valid when compiling a **C99** program or **C** program in which **stdbool.h** has been included.

Notes 9. When C89 is used for compiling, the size, number of bytes for alignment, and sign are the same as for the **unsigned long** type.

Notes 10. This data type does not include a concept of sign.

(2) Compound Type (C), Class Type (C++)

This section explains internal representation of array type, structure type, and union type data in C and class type data in C++.

Table 4.2 shows internal representation of compound type and class type data.

Table 4.2 Internal Representation of Compound Type and Class Type Data

Data Type	Alignment (bytes)	Size (bytes)	Data Allocation Example
Array	Array element alignment	Number of array elements × element size	`char a[10];` `Alignment:` 1 byte Size: 10 bytes

Structure	Maximum structure member alignment	Total size of members.
		Refer to (a) Structure Data Allocation, below.
			`struct {`
			`char a,b;`
			`};`
			Alignment: 1 byte
			Size: 2 bytes

Union	Maximum union member alignment	Maximum size of member.
		Refer to (b) Union Data Allocation, below.
			`union {`
			`char a,b;`
			`};`
			Alignment: 1 byte
			Size: 1 byte

Class	1. Always 4 if a virtual function is included
	2. Other than 1 above: maximum member alignment
		Sum of data members, pointer to the virtual function table, and pointer to the virtual base class.
		Refer to (c) Class Data Allocation, below.
			`class B:public A {`
			`virtual void f();`
			`};`
			Alignment: 4 bytes
			Size: 8 bytes
			`class A {`
			`char a;`
			`};`
			Alignment: 1 byte
			Size: 1 byte

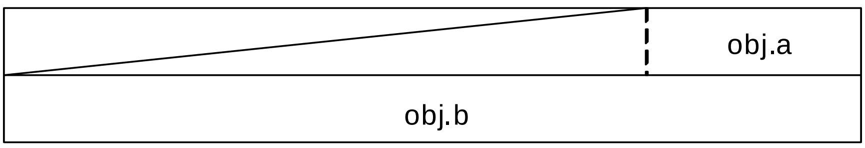
In the following examples, a rectangle ( ) indicates four bytes. The diagonal line ( ) represents an unused area for alignment. The address increments from right to left (the left side is located at a higher address).

(a) Structure Data Allocation

When structure members are allocated, an unused area may be generated between structure members to align them to boundaries.

Example

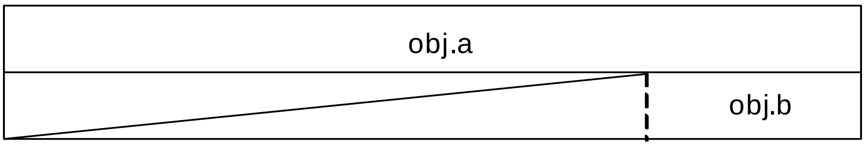
```
struct {
 char a;
 int b;
} obj
```



If a structure has 4-byte alignment and the last member ends at an 1-, 2-, or 3-byte address, the following three, two, or one byte is included in this structure.

Example

```
struct {
 int a;
 char b;
} obj
```

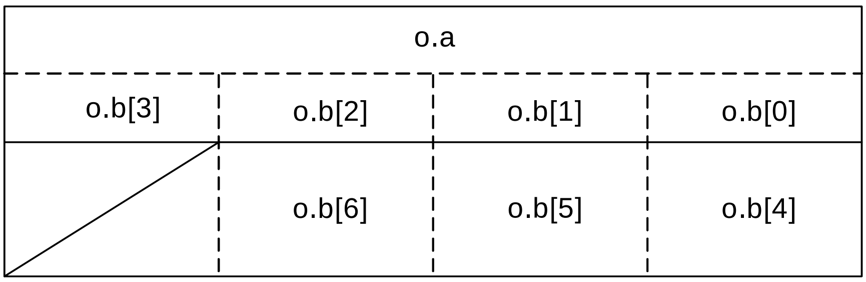


(b) Union Data Allocation

When an union has 4-byte alignment and its maximum member size is not a multiple of four, the remaining bytes up to a multiple of four is included in this union.

Example

```
union {
 int a;
 char b[7];
} o;
```

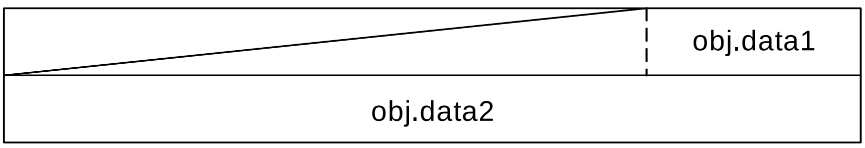


(c) Class Data Allocation

For classes having no base class or virtual functions, data members are allocated according to the allocation rules of structure data.

Example

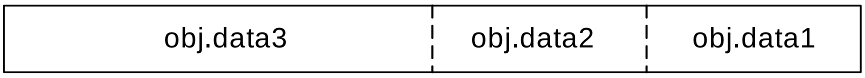
```
class A{
  char data1;
  int data2;
public:
  A();
  int getData1(){return data1;}
}obj;
```



If a class is derived from a base class of 1-byte alignment and the start member of the derived class is 1-byte data, data members are allocated without unused areas.

Example

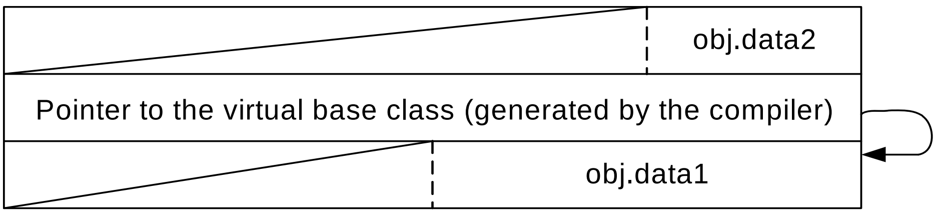
```
class A{
  char data1;
};
class B:public A{
  char data2;
  short data3;
}obj;
```



For a class having a virtual base class, a pointer to the virtual base class is allocated.

Example

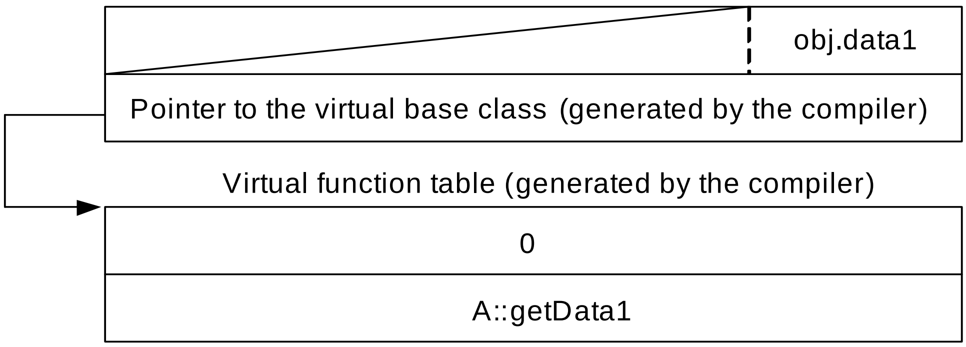
```
class A{
   short data1;
};
class B: virtual protected A{
   char data2;
}obj;
```



For a class having virtual functions, the compiler creates a virtual function table and allocates a pointer to the virtual function table.

Example

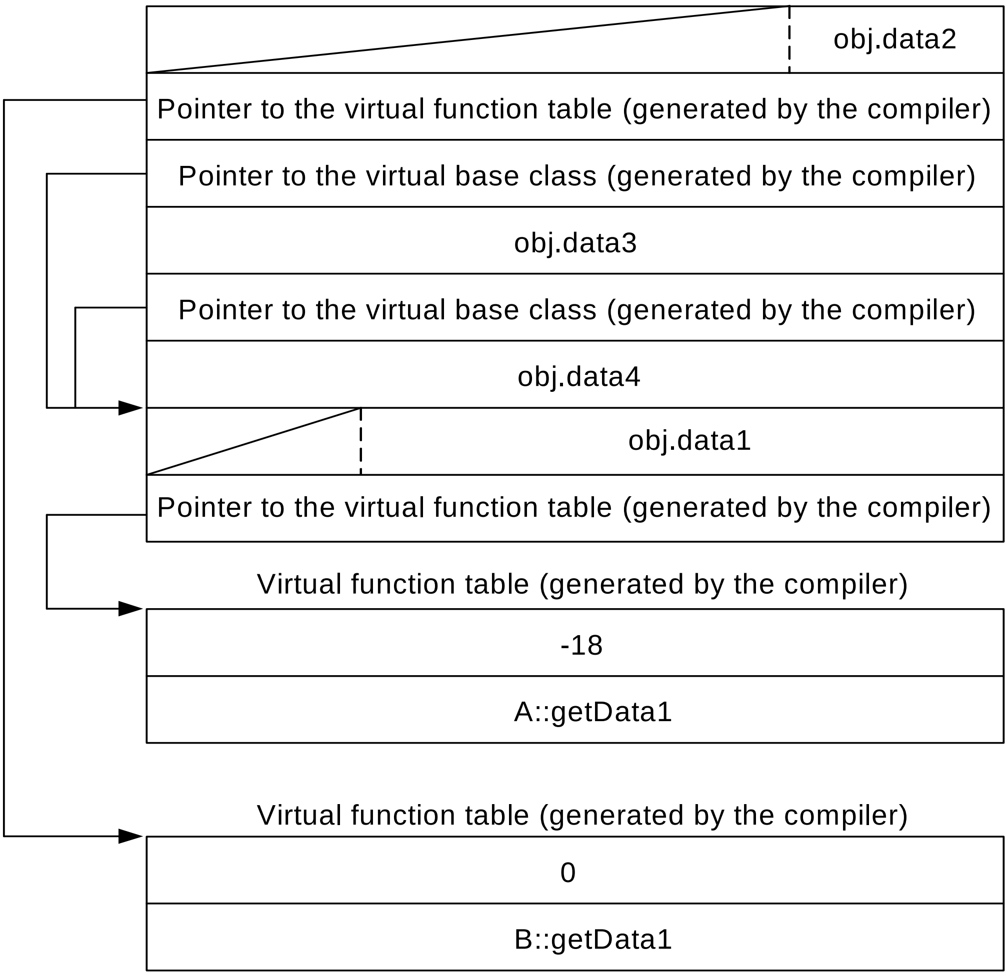
```
class A{
    char data1;
  public:
    virtual int getData1();
}obj;
```



An example is shown for class having virtual base class, base class, and virtual functions.

Example

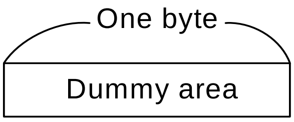
```
class A{
  char data1;
  virtual short getData1();
};
class B:virtual public A{
  char data2;
  char getData2();
  short getData1();
};
class C:virtual protected A{
  int data3;
};
class D:virtual public A,public B,public C{
  public:
  int data4;
  short getData1();
}obj;
```



For an empty class, a 1-byte dummy area is assigned.

Example

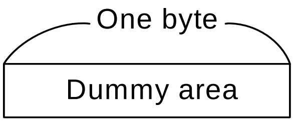
```
class A{
  void fun();
}obj;
```



For an empty class having an empty class as its base class, the dummy area is one byte.

Example

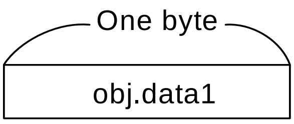
```
class A{
  void fun();
};
class B: A{
  void sub();
}obj;
```



Dummy areas shown in the above two examples are allocated only when the class size is 0. No dummy area is allocated if a base class or a derived class has a data member or has a virtual function.

Example

```
class A{
  void fun();
};
class B: A{
  char data1;
}obj;
```



(3) Bit Fields

A bit field is a member allocated with a specified size in a structure, a union, or a class. This section explains how bit fields are allocated.

(a) Bit Field Members

Table 4.3 shows the specifications of bit field members.

Table 4.3 Bit Field Member Specifications

No.	Item	Specifications
1	Type specifier allowed for bit fields	(unsigned)char, signed char, bool[*1], _Bool[*5], (unsigned)short, signed short, enum, (unsigned)int, signed int, (unsigned)long, signed long, (unsigned)long long, signed long long
2	How to treat a sign when data is extended to the declared type[*2]	Unsigned: Zero extension[*3] Signed: Sign extension[*4]
3	Sign type for the type without sign specification	Unsigned. When the **signed_bitfield** option is specified, the signed type is selected.
4	Sign type for **enum** type	Signed. When the **auto_enum** option is specified, the resultant type is selected.

Notes 1. The **bool** type is only valid for compilation of C++ programs or C programs including **stdbool.h**.

Notes 2. To use a bit field member, data in the bit field is extended to the declared type. One-bit field data declared with a sign is interpreted as the sign, and can only indicate 0 and −1.

Notes 3. Zero extension: Zeros are written to the upper bits to extend data.

Notes 4. Sign extension: The most significant bit of a bit field is used as a sign and the sign is written to the upper bits to extend data.

Notes 5. This data type is only valid for programs in C99. The **_Bool** type is treated as the **bool** type in compilation.

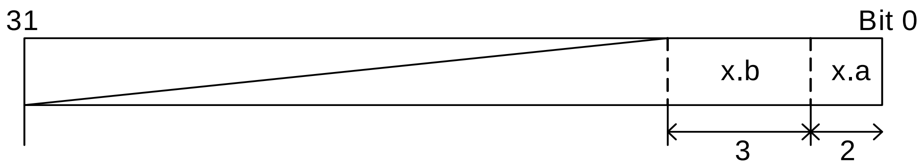
(b) Bit Field Allocation

Bit field members are allocated according to the following five rules:

- Bit field members are placed in an area beginning from the right, that is, the least significant bit.

Example

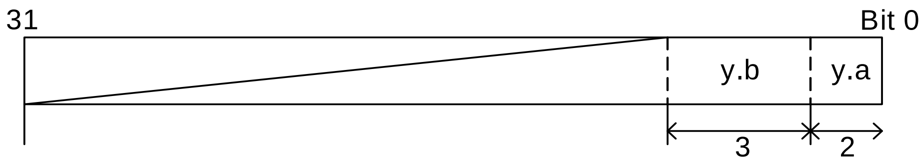
```
struct b1 {
  int a:2;
  int b:3;
} x;
```



- Consecutive bit field members having type specifiers of the same size are placed in the same area as much as possible.

Example

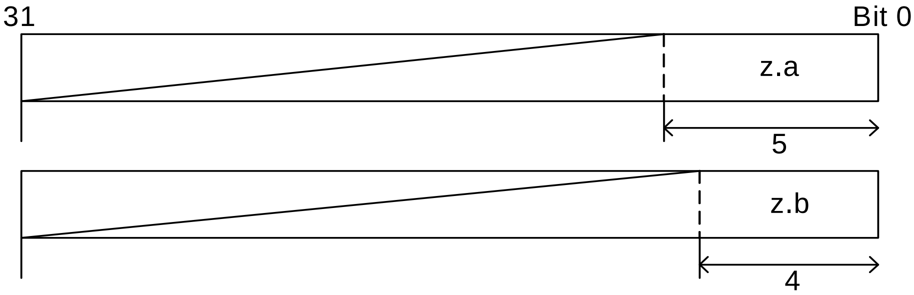
```
struct b1 {
  long          a:2;
  unsigned int  b:3;
} y;
```



- Bit field members having type specifiers with different sizes are allocated to separate areas.

Example

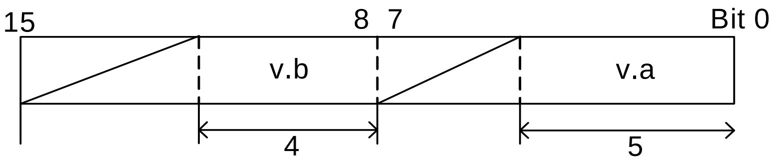
```
struct b1 {
  int    a:5;
  char   b:4;
} z;
```



- If the number of remaining bits in an area is less than the next bit field size, even though the type specifiers indicate the same size, the remaining area is not used and the next bit field is allocated to the next area.

Example

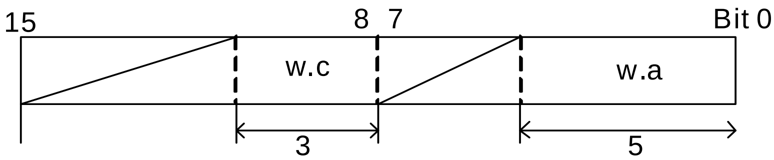
```
struct b2 {
  char   a:5;
  char   b:4;
} v;
```



- If a bit field member with a bit field size of 0 is declared, the next member is allocated to the next area.

Example

```
struct b2 {
  char   a:5;
  char    :0;
  char   c:3;
} w;
```



Note It is also possible to place bit field members from the upper bit. For details, refer to the description on the **bit_order** option in Compiler Options, and the description on **#pragma bit_order** in 4.2 Extended Language Specifications.

(4) Memory Allocation in Big Endian
In big endian, data are allocated in the memory as follows:

(a) One-Byte Data (char, signed char, unsigned char, bool[*1], and _Bool[*1] types)
The order of bits in one-byte data for the little endian and the big endian is the same.

Notes 1. When C89 is used for compiling, the size and the number of bytes for alignment are 4.

(b) Two-Byte Data ((signed) short and unsigned short types)
The upper byte and the lower byte will be reversed in two-byte data between the little endian and the big endian.

Example When two-byte data 0x1234 is allocated at address 0x100:

Little Endian: Address 0x100: 0x34	Big Endian: Address 0x100: 0x12
Address 0x101: 0x12	Address 0x101: 0x34

(c) Four-Byte Data ((signed) int[*2], unsigned int[*2], (signed) long, unsigned long, and float types)
The order of bytes will be reversed in four-byte data between the little endian and the big endian.

Notes 2. When the **int_to_short** option is specified, the **signed int** and **unsigned int** types have the same size and number of bytes for alignment as the **signed short** and **unsigned short** types, respectively.

Example When four-byte data 0x12345678 is allocated at address 0x100:

Little Endian:	Big Endian:
Address 0x100: 0x78	Address 0x100: 0x12
Address 0x101: 0x56	Address 0x101: 0x34
Address 0x102: 0x34	Address 0x102: 0x56
Address 0x103: 0x12	Address 0x103: 0x78

(d) Eight-Byte Data ((signed) long long, unsigned long long, and double types)
The order of bytes will be reversed in eight-byte data between the little endian and the big endian.

Example When eight-byte data 0x123456789abcdef is allocated at address 0x100:

Little Endian:	Big Endian:
Address 0x100: 0xef	Address 0x100: 0x01
Address 0x101: 0xcd	Address 0x101: 0x23
Address 0x102: 0xab	Address 0x102: 0x45
Address 0x103: 0x89	Address 0x103: 0x67
Address 0x104: 0x67	Address 0x104: 0x89
Address 0x105: 0x45	Address 0x105: 0xab
Address 0x106: 0x23	Address 0x106: 0xcd
Address 0x107: 0x01	Address 0x107: 0xef

(e) Compound-Type and Class-Type Data
Members of compound-type and class-type data will be allocated in the same way as that of the little endian. However, the order of byte data of each member will be reversed according to the rule of data size.

Example When the following function exists at address 0x100:

```
struct {
   short a;
   int b;
}z= {0x1234,  0x56789abc};
```

Little Endian:	Big Endian:
Address 0x100: 0x34	Address 0x100: 0x12
Address 0x101: 0x12	Address 0x101: 0x34
Address 0x102: Unused area	Address 0x102: Unused area
Address 0x103: Unused area	Address 0x103: Unused area
Address 0x104: 0xbc	Address 0x104: 0x56
Address 0x105: 0x9a	Address 0x105: 0x78
Address 0x106: 0x78	Address 0x106: 0x9a
Address 0x107: 0x56	Address 0x107: 0xbc

(f) Bit Field
Bit fields will be allocated in the same way as that of the little endian. However, the order of byte data in each area will be reversed according to the rule of data size.

Example When the following function exists at address 0x100:

```
struct {
   long a:16;
   unsigned int b:15;
   short c:5;
}y= {1,1,1};
```

Little Endian:	Big Endian:
Address 0x100: 0x01	Address 0x100: 0x00
Address 0x101: 0x00	Address 0x101: 0x01
Address 0x102: 0x01	Address 0x102: 0x00
Address 0x103: 0x00	Address 0x103: 0x01
Address 0x104: 0x01	Address 0x104: 0x00
Address 0x105: 0x00	Address 0x105: 0x01
Address 0x106: Unused area	Address 0x106: Unused area
Address 0x107: Unused area	Address 0x107: Unused area

(5) Floating-Point Number Specifications

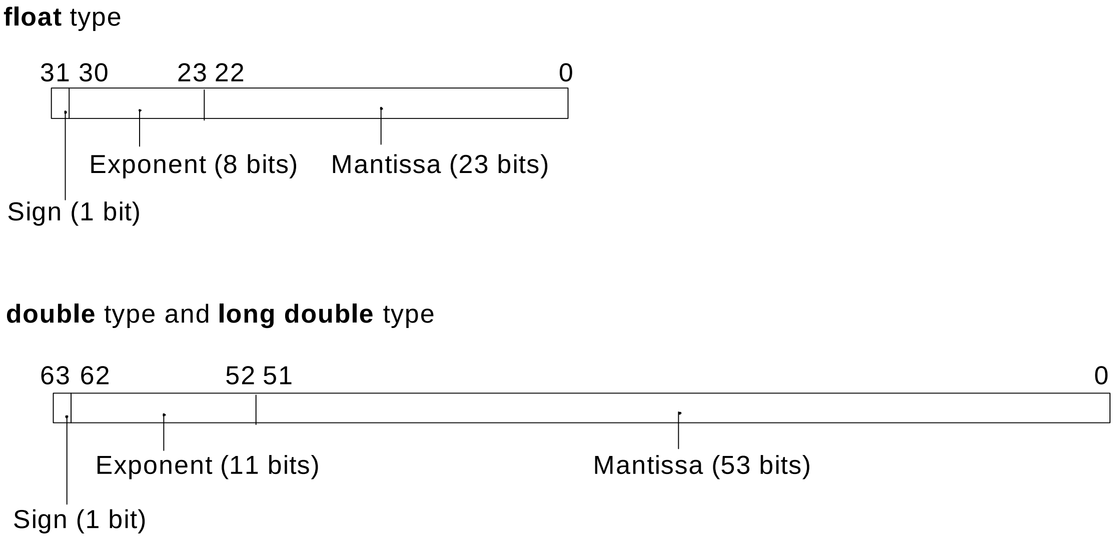
(a) Internal Representation of Floating-Point Numbers

Floating-point numbers handled by this compiler are internally represented in the standard IEEE format. This section outlines the internal representation of floating-point numbers in the IEEE format.
This section assumes that the **dbl_size=8** option is specified. When the **dbl_size=4** option is specified, the internal representation of the **double** type and **long double** type is the same as that of the **float** type.

(b) Format for Internal Representation
float types are represented in the IEEE single-precision (32-bit) format, while **double** types and **long double** types are represented in the IEEE double-precision (64-bit) format.

(c) Structure of Internal Representation
Figure 4.1 shows the structure of the internal representation of **float**, **double**, and **long double** types.

Figure 4.1 Structure of Internal Representation of Floating-Point Numbers



The internal representation format consists of the following parts:

i. Sign
Shows the sign of the floating-point number. 0 is positive, and 1 is negative.

ii. Exponent
Shows the exponent of the floating-point number as a power of 2.

iii. Mantissa
Shows the data corresponding to the significant digits (fraction) of the floating-point number.

(d) Types of Values Represented as Floating-Point Numbers
In addition to the normal real numbers, floating-point numbers can also represent values such as infinity. The following describes the types of values represented by floating-point numbers.

i. Normalized number
Represents a normal real value; the exponent is not 0 or not all bits are 1.

ii. Denormalized number
Represents a real value having a small absolute number; the exponent is 0 and the mantissa is other than 0.

iii. Zero
Represents the value 0.0; the exponent and mantissa are 0.

iv. Infinity
Represents infinity; all bits of the exponent are 1 and the mantissa is 0.

v. Not-a-number
Represents the result of operation such as "0.0/0.0", "∞/∞", or "∞–∞", which does not correspond to a number or infinity; all bits of the exponents are 1 and the mantissa is other than 0.
Table 4.4 shows the types of values represented as floating-point numbers.

Table 4.4 Types of Values Represented as Floating-Point Numbers

Mantissa	Exponent		
	0	Not 0 or Not All Bits are 1	All Bits are 1
0	0	Normalized number	Infinity
Other than 0	Denormalized number		Not-a-number

Note Denormalized numbers are floating-point numbers of small absolute values that are outside the range represented by normalized numbers. There are fewer valid digits in a denormalized number than in a normalized number. Therefore, if the result or intermediate result of a calculation is a denormalized number, the number of valid digits in the result cannot be guaranteed.
When **denormalize=off** is specified, denormalized numbers are processed as 0.
When **denormalize=on** is specified, denormalized numbers are processed as denormalized numbers.

(e) float Type

The **float** type is internally represented by a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa.

i. Normalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is between 1 and 254 ($2^8 - 2$). The actual exponent is gained by subtracting 127 from this value. The range is between –126 and 127. The mantissa is between 0 and $2^{23} - 1$. The actual mantissa is interpreted as the value of which 2^{23}rd bit is 1 and this bit is followed by the decimal point. Values of normalized numbers are as follows:

$(-1)^{sign} \times 2^{exponent-127} \times (1 + (mantissa) \times 2^{-23})$

Example

31	30 23	22 0
1	10000000	11000000000000000000000

Sign: –
Exponent: $10000000_{(2)} - 127 = 1$, where $_{(2)}$ indicates binary
Mantissa: $1.11_{(2)} = 1.75$
Value: $-1.75 \times 2^1 = -3.5$

ii. Denormalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is 0 and the actual exponent is –126. The mantissa is between 1 and $2^{23}-1$, and the actual mantissa is interpreted as the value of which 2^{23}rd bit is 0 and this bit is followed by the decimal point. Values of denormalized numbers are as follows:

$(-1)^{sign} \times 2^{-126} \times ((mantissa) \times 2^{-23})$

Example

31	30 23	22 0
0	00000000	11000000000000000000000

Sign: +
Exponent: –126
Mantissa: $0.11_{(2)} = 0.75$, where $_{(2)}$ indicates binary
Value: 0.75×2^{-126}

iii. Zero

The sign is 0 (positive) or 1 (negative), indicating +0.0 or –0.0, respectively. The exponent and mantissa are both 0.

+0.0 and –0.0 are both the value 0.0.

iv. Infinity

The sign is 0 (positive) or 1 (negative), indicating $+\infty$ or $-\infty$, respectively.

The exponent is 255 (2^8-1).

The mantissa is 0.

v. Not-a-number

The exponent is 255 (2^8-1).

The mantissa is a value other than 0.

Note A not-a-number is called a quiet NaN when the MSB of the mantissa is 1, or a signaling NaN when the MSB of the mantissa is 0. There are no stipulations regarding the values of the rest of the mantissa and of the sign.

(f) double Types and long double Types

The **double** and **long double** types are internally represented by a 1-bit sign, an 11-bit exponent, and a 52-bit mantissa.

i. Normalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is between 1 and 2046 ($2^{11}-2$). The actual exponent is gained by subtracting 1023 from this value. The range is between –1022 and 1023. The mantissa is between 0 and $2^{52}-1$. The actual mantissa is interpreted as the value of which 2^{52}nd bit is 1 and this bit is followed by the decimal point. Values of normalized numbers are as follows:

$(-1)^{sign} \times 2^{exponent-1023} \times (1+(mantissa) \times 2^{-52})$

Example

63	62 52	51 0
0	01111111111	1110000000000000000000000000000000000000000000000000

Sign: +
Exponent: $1111111111_{(2)} - 1023 = 0$, where $_{(2)}$ indicates binary

Mantissa: $1.111_{(2)} = 1.875$
Value: $1.875 \times 2^{0} = 1.875$

ii. Denormalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is 0 and the actual exponent is –1022. The mantissa is between 1 and 2^{52}–1, and the actual mantissa is interpreted as the value of which 2^{52}nd bit is 0 and this bit is followed by the decimal point. Values of denormalized numbers are as follows:

$(-1)^{sign} \times 2^{-1022} \times ((mantissa) \times 2^{-52})$

Example

63 62 52 51 0

1	00000000000	1110000000000000000000000000000000000000000000000000

Sign: –
Exponent: –1022
Mantissa: $0.111_{(2)}$ = 0.875, where $_{(2)}$ indicates binary
Value: 0.875×2^{-1022}

iii. Zero

The sign is 0 (positive) or 1 (negative), indicating +0.0 or –0.0, respectively. The exponent and mantissa are both 0.
+0.0 and –0.0 are both the value 0.0.

iv. Infinity

The sign is 0 (positive) or 1 (negative), indicating $+\infty$ or $-\infty$, respectively. The exponent is 2047 (2^{11}–1).
The mantissa is 0.

v. Not-a-number

The exponent is 2047 (2^{11}–1).
The mantissa is a value other than 0.

Note A not-a-number is called a quiet NaN when the MSB of the mantissa is 1, or signaling NaN when the MSB of the mantissa is 0. There are no specifications regarding the values of other mantissa fields or the sign.

4.1.6 Operator Evaluation Order

When an expression includes multiple operators, the evaluation order of these operators is determined according to the precedence and the associativity indicated by right or left.

Table 4.5 shows each operator precedence and associativity.

Table 4.5 Operator Precedence and Associativity

Precedence	Operators	Associativity	Applicable Expression
1	++ -- (postfix) ( ) [ ] -> .	Left	Postfix expression
2	++ -- (prefix) ! ~ + - * & sizeof	Right	Unary expression
3	(Type name)	Right	Cast expression
4	* / %	Left	Multiplicative expression
5	+ -	Left	Additive expression
6	<< >>	Left	Bitwise shift expression
7	< <= > >=	Left	Relational expression
8	== !=	Left	Equality expression
9	&	Left	Bitwise AND expression
10	^	Left	Bitwise exclusive OR expression
11	|	Left	Bitwise inclusive OR expression

Precedence	Operators	Associativity	Applicable Expression
12	&&	Left	Logical AND operation
13	||	Left	Logical inclusive OR expression
14	?:	Right	Conditional expression
15	= += -= *= /= %= <<= >>= &= |= ^=	Right	Assignment expression
16	,	Left	Comma expression

4.1.7 Conforming Language Specifications

(1) C Language Specifications (When the lang=c Option is Selected)
ANSI/ISO 9899-1990 American National Standard for Programming Languages -C

(2) C Language Specifications (When the lang=c99 Option is Selected)
ISO/IEC 9899:1999 INTERNATIONAL STANDARD Programming Languages - C

(3) C++ Language Specifications (When the lang=cpp Option is Selected)
Based on the language specifications compatible with Microsoft® Visual C/C++ 6.0

4.2 Extended Language Specifications

This section explains the extended language specifications supported by the CCRX.
The compiler supports the following extended specifications:

- **#pragma** extension specifiers and keywords
- Intrinsic functions
- Section address operators

4.2.1 Macro Names

The following shows supported macro names.

Table 4.6 Predefined Macros of Compiler

No.	Macro Name	Value	Option
1	__DATE__	Date of translating source file (character string constant in the form of "Mmm dd yyyy".) Here, the name of the month is the same as that created by the asctime function stipulated by ANSI standards (3 alphabetic characters with only the first character is capital letter) (The first character of dd is blank if its value is less than 10).	-
2	__FILE__	Name of assumed source file (character string constant).	-
3	__LINE__	Line number of source line at that point (decimal).	-
4	__STDC__	1	-
5	__STDC_HOSTED__	1	lang=c99
6	__STDC_VERSION__	199409L (lang = c*1) 199901L (lang = c99)	lang=c*1 lang=c99
7	__STDC_IEC_559__	1	lang=c99
8	__STDC_IEC_559_COMPLEX__	1	lang=c99
9	__STDC_ISO_10646__	199712L	lang=c99
10	__cplusplus	1	lang=cpp*2 lang=ecpp
11	__TIME__	Translation time of source file (character string constant having format "hh:mm:ss").	-
12	#define __RXV1 #define __RXV2 #define __RXV3 [V3.00.00 or later]	1 1 1	isa=rxv1 *3 isa=rxv2 *3 isa=rxv3 *3
13	#define __BIG #define __LIT	1 1	endian=big endian=little
14	#define __DBL4 #define __DBL8	1 1	dbl_size=4 dbl_size=8
15	#define __INT_SHORT	1	int_to_short

No.	Macro Name	Value	Option
16	#define __SCHAR #define __UCHAR	1 1	signed_char unsigned_char
17	#define __SBIT #define __UBIT	1 1	signed_bitfield unsigned_bitfield
18	#define __ROZ #define __RON	1 1	round=zero round=nearest
19	#define __DOFF #define __DON	1 1	denormalize=off denormalize=on
20	#define __BITLEFT #define __BITRIGHT	1 1	bit_order=left bit_order=right
21	#define __AUTO_ENUM	1	auto_enum
22	#define __FUNCTION_LIB #define __INTRINSIC_LIB	1 1	library=function library=intrinsic
23	#define __FPU	1	fpu
24	#define __RENESAS__ *4	1	-
25	#define __RENESAS_VERSION__ *4	0x*XXYYZZ*00 *5	-
26	#define __RX *4	1	-
27	#define __PIC	1	pic
28	#define __PID	1	pid
29	#define __RX600 #define __RX200	1 1	cpu=rx600 cpu=rx200
30	#define __CCRX__ *4 [V2.03.00 or later]	1	-
31	#define __RX_ISA_VERSION__ [V3.00.00 or later]	1 2 3	isa=rxv1 *3 isa=rxv2 *3 isa=rxv3 *3
32	#define __DPFPU [V3.01.00 or later]	1	dpfpu
33	#define __TFU [V3.01.00 or later]	1 2 (In the case of tfu_version=v2)	tfu=intrinsic tfu=intrinsic,mathlib
34	#define __TFU_MATHLIB [V3.01.00 or later]	1 2 (In the case of tfu_version=v2)	tfu=intrinsic,mathlib

Notes 1. Includes cases where a file with the **.c** extension is compiled without specifying the **-lang** option.

Notes 2. Includes cases where a file with the **.cpp**, **.cp**, or **.cc** extension is compiled without specifying the **-lang** option.

Notes 3. Includes the specification by the ISA_RX environment variable.

Notes 4. Always defined regardless of the option.

Notes 5. When the Compiler version is V*XX.YY.ZZ*, the value of **__RENESAS_VERSION__** is 0x*XXYYZZ*00.
Example
For V3.01.00: **#define __RENESAS_VERSION__ 0x03010000**

4.2.2 Keywords

The CCRX adds the following characters as a keyword to implement the extended function. These words are similar to the ANSI C keywords, and cannot be used as a label or variable name.

Keywords that are added by the CCRX are listed below.

__evenaccess, far, _far, near, and _near

Table 4.7 Keywords

No.	Keyword	Function
1	#pragma STDC CX_LIMITED_RANGE #pragma STDC FENV_ACCESS #pragma STDC FP_CONTRACT	Reserved keywords that are only valid when C99 is selected (these are only for grammatical checking and not for checking the correctness of the code).
2	#pragma keywords	Provides language extensions. For details, refer to 4.2.3 #pragma Directive.
3	__evenaccess	Guarantees access in the size of the variable type.
4	far _far near _near	Reserved keywords (these are ignored even though they are recognized as type names)
5	_RAM_BASE	Reserved keyword. (Only in -base=ram sepcified.)
6	_ROM_BASE	Reserved keyword. (Only in -base=rom sepcified.)
7	_PID_TOP	Reserved keyword. (Only in -pid sepcified.)
8	_builtin_xxx	Reserved keyword. This means all functions those names begin with _builtin_.

4.2.3 #pragma Directive

The following extended specifications can also be used with the _Pragma operator in C99.

(1) Section Switch

This extension changes the section name to be output by the compiler.

For details on the description, refer to(1) Section Switch.

```
#pragma section [<section type>] [D<new section name>]
<section type>: { P | C | D | B }
```

(2) Stack Section Creation

This extension creates the stack section.

For details on the description, refer to (2) Stack Section Creation.

```
#pragma stacksize { si=<constant> | su=<constant> }
```

(3) Interrupt Function Creation

This extension creates the interrupt function.

For details on the description, refer to (3) Interrupt Function Creation.

```
#pragma interrupt [(]<function name>[(<interrupt specification>[,...])][,...][)]
```

(4) Inline Expansion of Function

This extension expands a function.

For details on the description, refer to (4) Inline Expansion of Function.

```
#pragma inline [(]<function name>[,...][)]
```

(5) Cancellation of Inline Expansion of Function
This extension cancels expansion of a function.
For details on the description, refer to (4) Inline Expansion of Function.

```
#pragma noinline [(]<function name>[,...][)]
```

(6) Inline Expansion of Assembly-Language Function
This extension creates the assembly-language inline functions.
For details on the description, refer to (5) Inline Expansion of Assembly-Language Function.

```
#pragma inline_asm[(]<function name>[,...][)]
```

(7) Entry Function Specification
This extension specifies the entry function.
For details on the description, refer to (6) Entry Function Specification.

```
#pragma entry[(]<function name>[)]
```

(8) Bit Field Order Specification
This extension specifies the order of the bit field.
For details on the description, refer to (7) Bit Field Order Specification.

```
#pragma bit_order [{left | right}]
```

(9) 1-Byte Alignment Specification for Structure Members and Class Members
This extension specifies the boundary alignment value of structure members and class members as 1 byte.
For details on the description, refer to (8) Alignment Value Specification for Structure Members and Class Members.

```
#pragma pack
```

(10) Default Alignment Specification for Structure Members and Class Members
This extension specifies the boundary alignment value for structure members and class members as the value for members.
For details on the description, refer to (8) Alignment Value Specification for Structure Members and Class Members.

```
#pragma unpack
```

(11) Option Alignment Specification for Structure Members and Class Members
This extension specifies the option of the boundary alignment value for structure members and class members.
For details on the description, refer to (8) Alignment Value Specification for Structure Members and Class Members.

```
#pragma packoption
```

(12) Allocation of a Variable to the Absolute Address
This extension allocates the specified variable to the specified address.
For details on the description, refer to (9) Allocation of a Variable to the Absolute Address.

```
#pragma address [(]<variable name>=<absolute address>[,...][)]
```

(13) Endian Specification for Initial Values
This extension specifies an endian for initial values.
For details on the description, refer to (10) Endian Specification for Initial Values.

```
#pragma endian [{big | little}]
```

(14) Specification of Function in which Instructions at Branch Destinations are Aligned to 4-Byte Boundaries
This extension specifies the function in which instructions at branch destinations are aligned to 4-byte boundaries.

For details on the description, refer to (11) Specification of Function in which Instructions at Branch Destinations are Aligned for Execution.

```
#pragma instalign4 [(]<function name>[(<branch destination type>)][,...][)]
```

(15) Specification of Function in which Instructions at Branch Destinations are Aligned to 8-Byte Boundaries
This extension specifies the function in which instructions at branch destinations are aligned to 8-byte boundaries.
For details on the description, refer to (11) Specification of Function in which Instructions at Branch Destinations are Aligned for Execution.

```
#pragma instalign8 [(]<function name>[(<branch destination type>)][,...][)]
```

(16) Specification of Function in which Instructions at Branch Destinations are not Aligned
This extension specifies the function in which instructions at branch destinations are not aligned.
For details on the description, refer to (11) Specification of Function in which Instructions at Branch Destinations are Aligned for Execution.

```
#pragma noinstalign [(]<function name>[,...][)]
```

(17) Specification of Function for generating a code for detection of stack smashing [Professional Edition only] [V2.04.00 or later]
This extension generates a code for detection of stack smashing.
For details on the description, refer to (12) Specification of Function for generating a code for detection of stack smashing [Professional Edition only] [V2.04.00 or later].

```
#pragma stack_protector [(] function name [(num=<integer value>)] [)]
```

(18) Specification of not generating a code for detection of stack smashing [Professional Edition only] [V2.04.00 or later]
This extension suppress generating generate a code for detection of stack smashing.
For details on the description, refer to (12) Specification of Function for generating a code for detection of stack smashing [Professional Edition only] [V2.04.00 or later].

```
#pragma no_stack_protector [(] function name [)]
```

4.2.4 Using Extended Specifications

This section explains using the following extended specifications.

- Section switch
- Stack section creation
- Interrupt function creation
- Inline expansion of function
- Inline expansion of assembly-language function
- Entry function specification
- Bit field order specification
- Alignment value specification for structure members and class members
- Allocation of a variable to the absolute address
- Endian specification for initial values
- Specification of function in which instructions at branch destinations are aligned for execution
- Specification of function for generating a code for detection of stack smashing

(1) Section Switch

```
#pragma section [<section type>] [D<new section name>]
<section type>: { P | C | D | B }
```

This extension changes the section name to be output by the compiler.
When both a section type and a new section name are specified, the section names for all functions written after the **#pragma** declaration are changed if the specified section type is **P**. If the section type is **C**, **D**, or **B**, the names of all sections defined after the **#pragma** declaration are changed.
When only a new section name is specified, the section names for the program, constant, initialized data, and uninitialized data areas after the **#pragma** declaration are changed. In this case, the default section name post-fixed with the string specified by <new section name> is used as the new section name.
When neither a section type nor a new section name is specified, the section names for the program, constant, initialized data, and uninitialized data areas after the **#pragma** declaration are restored to the default section names. The default section name for each section type is determined by the **section** option when specified. If the default section name is not specified by the **section** option, the section type name is used instead.
[Earlier than V3.02.00] **#pragma section** must be specified outside function definitions.
[V3.02.00 or later] **#pragma section** can be specified within function definitions. The opening and closing parentheses of a function do not affect the valid range of **#pragma**.
[V3.02.00 or later] **#pragma section** has effect on the name of the section containing static variables within the function, static data members, and their initial values. However, there is no effect on the names of the following sections:

- Section that contains static variables within the function and their initial values specified in a function template (not specialized)
- Section that contains static data members and their initial values specified in a class template

Examples 1. When a section name and a section type are specified

```
#pragma section B Ba
int i; // Allocated to the Ba section
void func(void)
{
(omitted)
}

#pragma section B Bb
int j;// Allocated to the Bb section
void sub(void)
{
(omitted)
}
```

Examples 2. When the section type is omitted

```
#pragma section abc
int a; // Allocated to the Babc section
const int c=1; // Allocated to the Cabc section
void f(void)// Allocated to the Pabc section
{
    a=c;
}

#pragma section
int b;// Allocated to the B section
void g(void)// Allocated to the P section
{
    b=c;
}
```

Examples 3. Specification for static class members

```
/*
** Class member declaration
*/

class A {
  private:
    // No initial value specified
#pragma section DATA
    static int data_;
#pragma section
    // Initial value specified
#pragma section TABLE
    static int table_[2];
#pragma section
};

/*
** Class member definition
*/

// No initial value specified
#pragma  section  DATA
int A::data_;
#pragma section

// Initial value specified
#pragma section TABLE
int A::table_[2] = { 0,  1 };
#pragma section
```

Examples 4. Specification of a section for static variables within the function [V3.02.00 or later]

```
oid test1(void) {
#pragma section B B1
  static int b1;  // B1
#pragma section B B2
  static int b2;  // B2
}

// The valid range is not affected by the parentheses of the function definition.
int b3;  // B2

void test2(void) {
  static int b4;  // B2
#pragma section
  static int b5;  // B
}
```

The section name of the following items cannot be changed by this extension. The **section** option needs to be used.
(1) String literal and initializers for use in the dynamic initialization of aggregates
(2) Branch table of **switch** statement
Up to 2045 sections can be specified by **#pragma section** in one file.
When specifying the section for static class member variables, be sure to specify **#pragma section** for both the class member declaration and definition.

Example

```
/*
 ** Class member declaration
 */

class A
{
private:

// No initial value specified
#pragma section DATA
static int data_;
#pragma section

// Initial value specified
#pragma section TABLE
static int table_[2];
#pragma section
};

/*
** Class member definition
*/

// No initial value specified
#pragma section DATA
int A::data_;
#pragma section

// Initial value specified
#pragma section TABLE
int A::table_[2]={0, 1};
#pragma section
```

(2) Stack Section Creation

```
#pragma stacksize { si=<constant> | su=<constant> }
```

When si=<constant> is specified, a data section is created to be used as the stack of size <constant> with section name SI.
When su=<constant> is specified, a data section is created to be used as the stack of size <constant> with section name SU.
C source description:

```
#pragma stacksize si=100
#pragma stacksize su=200
```

Example of expanded code:

```
.SECTION    SI,DATA,ALIGN=4
.BLKB       100
.SECTION    SU,DATA,ALIGN=4
.BLKB       200
```

si and **su** can each be specified only once in a file.
<constant> must always be specified as a multiple of four.
A value from 4 to 2147483644(0x7ffffffc) is specifiable for <constant>.

(3) Interrupt Function Creation

```
#pragma interrupt [(]<function name>[(<interrupt specification>[,...])][,...][)]
```

This extension declares an interrupt function.
A global function or a static function member can be specified for the function name.
Table 4.8 lists the interrupt specifications.

Table 4.8 Interrupt Specifications

No.	Item	Form	Options	Specifications
1	Vector table	vect=	<vector number>	Specifies the vector number for which the interrupt function address is stored.
2	Fast interrupt	fint	None	Specifies the function used for fast interrupts. This **RTFI** instruction is used to return from the function.
3	Limitation on registers in interrupt function	save	None	Limits the number of registers used in the interrupt function to reduce save and restore operations.
4	Nested interrupt enable	enable	None	Sets the **I** flag in **PSW** to 1 at the beginning of the function to enable nested interrupts.
5	Accumulator saving	acc	None	Saves and restores Accumulator in the interrupt function.
6	Accumulator non-saving	no_acc	None	Does not save and restore Accumulator in the interrupt function.
7	Enable the register bank save function [V3.01.00 or later]	bank=	<bank number>	Enables the register bank save function. Specify the number of the bank in which the values of registers will be saved.
8	Saving output of the trigonometric function unit [V3.05.00 or later]	tfu	None	Saves and restores the output of the trigonometric function unit in the interrupt function.
9	Not saving output of the trigonometric function unit [V3.05.00 or later]	no_tfu	None	Does not save or restore output of the trigonometric function unit in the interrupt function.

An interrupt function declared by **#pragma interrupt** guarantees register values before and after processing (all registers used by the function are pushed onto and popped from the stack when entering and exiting the function). The **RTE** instruction directs execution to return from the function in most cases.
An interrupt function with no interrupt specifications is processed as a simple interrupt function.

When use of the vector table is specified (**vect=**), the interrupt function address is stored in the specified vector table number location in the C$VECT section.
[V3.00.00 or later] When the **-split_vect** option is specified in the optimizing linkage editor, the C$VECT section is split by vector table number and each section has the name of "C$VECT<vector table number>".

When use of fast interrupt processing is specified (**fint**), the **RTFI** instruction is used to return from the function.
When the **fint_register** option is also specified, the registers specified through the option are used by the interrupt function without being saved or restored.

When a limitation on registers in interrupt function is specified (**save**), the registers that can be used in the interrupt function are limited to R1 to R5 and R14 to R15. R6 to R13 are not used and the instructions for saving and restoring them are not generated.

When **enable** is specified, the **I** flag in **PSW** is set to 1 at the beginning of the function to enable nested interrupts.

When Accumulator saving (**acc**) is specified, if another function is called from a function specified with **#pragma interrupt** or the function uses an instruction that modifies the **ACC**, an instruction to save and restore the **ACC** is generated. When RXv1 is selected as ISA*1, the **ACC** is saved and restored. When a value other than RXv1 is selected as ISA*1, **ACC0** and **ACC1** are saved and restored.

When Accumulator non-saving is specified (**no_acc**), an instruction to save and restore the **ACC** is not generated. If neither **acc** nor **no_acc** is specified, the result depends on the setting of the compiler option **-save_acc**.

[V3.01.00 or later] When **bank = <bank number>** is specified, the CC-RX enables the use of the register bank save function (for which the SAVE and RSTR instructions are provided) in cases where it is considered necessary. The destination for saving data is the save register bank with the number specified in **<bank number>**. Refer to the *User's Manual: Hardware* for the target MCU and specify a bank number that is actually available.
When this facility is enabled, the values of **ACC0** and **ACC1** are also saved and restored even if the **-save_acc** option is not specified or **no_acc** (not saving accumulator values) is specified. Specifying **bank = <bank number>** when RXv1 or RXv2 is selected as ISA*1 will lead to an error. When you specify **bank = <bank number>**, also specify the assembler option **-bank**. If you are using an integrated development environment from Renesas, **-bank** is specified automatically upon selection of an MCU that has save register banks.

[V3.05.00 or later] When **tfu** (saving output of the trigonometric function unit) is specified, if another function is called from a function specified with **#pragma interrupt** or the function uses the trigonometric function unit, instructions to save and restore the output (DTSR0, DTSR1) of the trigonometric function unit are generated.
When **no_tfu** (not saving output of the trigonometric function unit) is specified, instructions to save and restore output of the trigonometric function unit are not generated.
Interrupt specifications **tfu** and **no_tfu** can be specified only when **tfu_version=v2** is specified. If neither **tfu** nor **no_tfu** is specified, the result depends on the setting of the **-nosave_tfu** compile option.

Note *1) This means a selection by the **-isa** option or the ISA_RX environment variable.

The function must return only **void** data. No return value can be specified for the **return** statement. If attempted, an error will be output.

Examples 1. Correct declaration and wrong declaration

```
#pragma interrupt (f1, f2)
void f1(){...}// Correct declaration.
int f2(){...}// An error will be output
// because the return value is not
// void data.
```

Examples 2. General interrupt function

C source description:

```
#pragma interrupt func
void func(){ .... }
```

Output code:

```
_func:
PUSHM R1-R3; Saves the registers used in the function.
....
(R1, R2, and R3 are used in the function)
....
POPM R1-R3; Restores the registers saved at the entry.
RTE
```

Examples 3. Interrupt function that calls another function

In addition to the registers used in the interrupt function, the registers that are not guaranteed before and after a function call are also saved at the entry and restored at the exit.

C source description:

```
#pragma interrupt func
void func(){
  ...
  sub();
  ...
}
```

Output code:

```
_func:
    PUSHM R14-R15
    PUSHM R1-R5
    ...
    BSR _sub
    ...
    POPM R1-R5
    POPM R14-R15
    RTE
```

Examples 4. Use of interrupt specification fint

C source description: Compiles with the fint_register=2 option specified

```
#pragma interrupt func1(fint)
void func1(){ a=1; }    //  Interrupt function
void func2(){ a=2; }    //  General function
```

Output code:

```
_func1:
    PUSHM R1-R3 ;  Saves the registers used in the function.
    ...         ; (Note that R12 and R13 are not saved.)
    ...
    (R1, R2, R3, R12, and R13 are used in the function.)
    ...
    POPM R1-R3  ;  Restores the registers saved at the entry.
    RTFI

_func2:
    ...         ; In the functions without #pragma interrupt fint
    ...         ; specification, do not use R12 and R13.
    RTE
```

Examples 5. Use of interrupt specification acc

C source description:

```
void func5(void);
#pragma interrupt accsaved_ih(acc)      /* Specifies acc */
void accsaved_ih(void)
{
     func5();
}
```

Output code:

```
_accsaved_ih:
     PUSHM R14-R15
     PUSHM R1-R5
     MVFACMI R4
     SHLL #10H, R4
     MVFACHI R5
     PUSHM R4-R5
     BSR _func5
     POPM R4-R5
     MVTACLO R4
     MVTACHI R5
     POPM R1-R5
     POPM R14-R15
     RTE
```

[Remarks]
Due to the specifications of the RX instruction set, if rxv1 is specified for ISA[*1], only the upper 48 bits of **ACC** can be saved and restored with the **acc** flag. The lower 16 bits of **ACC** are not saved or restored.
Each interrupt specification can be specified only with alphabetical lowercase letters. When specified with uppercase letters, an error will occur.
When **vect** is used as an interrupt specification, the address of empty vectors for which there is no specification is 0. You can specify a desired address value or symbol for an address with the optimizing linkage editor. For details, refer to the descriptions on the **VECT** and **VECTN** options.
Parameters are not definable for **#pragma interrupt** functions. Although defining parameters for such functions does not lead to an error, values read out from the parameters are undefined.

Note *1) ISA indicates the selection by using the **isa** option or ISA_RX environment variable.

Purpose of **acc** and **no_acc**:
acc and **no_acc** take into account the following purposes:

- Solution for decrease in the interrupt response speed when compensation of ACC is performed by **save_acc** (no_acc)
  Though the save_acc option is valid for compensation of ACC in an existing interrupt function, the interrupt response speed is degraded in some cases. Therefore, **no_acc** is provided as a means to disable saving and restoring of unnecessary ACC for each function independently.
- Control of saving and restoring of ACC through source code
  Explicitly selecting acc or no_acc for an interrupt function for which saving and restoring of ACC has already been considered allows saving and restoring of ACC to be defined in the source program without using the save_acc option.

Examples 6. Use of interrupt specification **bank**
C source description:

```
#pragma interrupt func(bank=3)      /* Specifies "bank=3" */
void func(void)
{
     ...
}
```

Output code:

```
_func:
     SAVE #03H ;  Instruction for saving values in save register bank 3
     ...
     RSTR #03H ;  Instruction for restoring values from save register bank 3
     RTE
```

Examples 7. Use of interrupt specification **vect** and **bank** in combination
C source description:

```
#pragma interrupt func(vect=64, bank=4) /* Specifies "vect=64" and "bank=4" */
void func(void)
{
     ...
}
```

Output code:

```
_func:
     .RVECTOR      64,_func ;  Vector table number registration
     SAVE #04H ;  Instruction for saving values in save register bank 4
     ...
     RSTR #04H ;  Instruction for restoring values from save register bank 4
     RTE
```

(4) Inline Expansion of Function

```
#pragma inline [(]<function name>[,...][)
#pragma noinline [(]<function name>[,...][)]
```

#pragma inline declares a function for which inline expansion is performed.
Even when the **noinline** option is specified, inline expansion is done for the function specified by **#pragma inline**.
#pragma noinline declares a function for which the inline option effect is canceled.
A global function or a static function member can be specified as a function name.
When inline expansion is performed for a function specified by **#pragma inline** or a function with the **inline** function specifier (C++ and C (C99), the body of the function is expanded where the function is called.

Example Source file:

```
#pragma inline(func)
static int func (int a, int b)
{
    return (a+b)/2;
}
int x;
main()
{
    x=func(10,20);
}
```

Inline expansion image:

```
int x;
main()
{
    int func_result;
    {
        int a_1=10, b_1=20;
        func_result=(a_1+b_1)/2;
    }
    x=func_result;
}
```

Inline expansion will not be applied in the following functions even when **#pragma inline** is specified.

- The function has variable parameters.
- Another function is called by using the address of the function to be expanded.

#pragma inline does not guarantee inline expansion; inline expansion might not be applied due to restrictions on increasing compilation time or memory size. If inline expansion is canceled, try specifying the **noscope** option; inline expansion may be applied in some cases.
Specify **#pragma inline** before defining a function.
An external definition is generated for a function specified by **#pragma inline**.
When **#pragma inline** is specified for a **static** function, the function definition is deleted after inline expansion.
The C++ compiler does not create external definitions for inline-specified functions.
The C (C99) does not create external definitions for inline-specified functions unless they include **extern** declarations.

(5) Inline Expansion of Assembly-Language Function

```
#pragma inline_asm[(]<function name>[,...][)]
```

This extension declares an assembly-language function for which inline expansion is performed.
The general function calling rules are also applied to the calls of assembly-language inline functions.

Example C source description:

```
#pragma inline_asm func
static int func(int a, int b){
ADD R2,R1; Assembly-language description
}
main(int *p){
*p = func(10,20);
}
```

Output code:

```
_main:
   PUSH.L R6
   MOV.L R1, R6
   MOV.L #20, R2
   MOV.L #10, R1
   ADD  R2,R1; Assembly-language description
   MOV.L R1, [R6]
   MOV.L #0, R1
   RTSD #04H, R6-R6
```

Specify **#pragma inline_asm** before defining a function.
An external definition is generated for a function which is not a **static** function but for which **#pragma inline_asm** is specified.
When the registers whose values are saved and restored at the entry and exit of a function (see Table 9.1 Rules to Use Registers) are used in an assembly-language inline function, these registers must be saved and restored at the start and end of the function.

[Remarks]

- In an assembly-language inline function, use only the RX Family instruction and temporary labels. Other labels cannot be defined and assembler directives cannot be used.
- Do not use **RTS** at the end of an assembly-language inline function.
- Function members cannot be specified as function names.
- When **#pragma inline_asm** is specified for a **static** function, the function definition is deleted after inline expansion.
- Assembly-language descriptions are processed by the preprocessor; take special care when defining through **#define** a macro with the same name as an instruction or a register used in the assembly language (such as **MOV** or **R5**).

- A stack information file handles the assembly code for a **#pragma inline_asm** directive as not consuming stack area. Be careful when the assembly code includes an instruction with **R0** as an operand.

(6) Entry Function Specification

```
#pragma entry[(]<function name>[)]
```

This specifies that the function specified as <function name> is handled as an entry function.
The entry function is created without any code to save and restore the contents of registers.
When **#pragma stacksize** is declared, the code that makes the initial setting of the stack pointer will be output at the beginning of the function.
When the **base** option is specified, the base register specified by the option is set up.

Example C source description: **-base=rom=R13** is specified

```
#pragma stacksize su=100
#pragma entry INIT
void INIT() {
:
}
```

Output code:

```
.SECTION    SU,DATA,ALIGN=4
.BLKB       100
.SECTION    P,CODE
_INIT:
MVTC        (TOPOF SU + SIZEOF SU),USP
MOV.L       #__ROM_TOP,R13
```

Be sure to specify **#pragma entry** before declaring a function.
Do not specify more than one entry function in a load module.

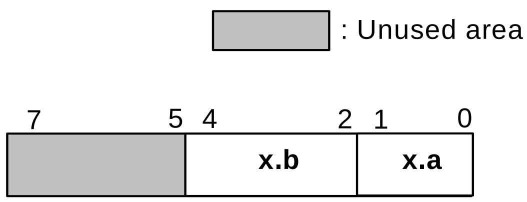
(7) Bit Field Order Specification

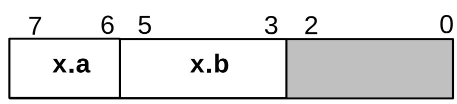
```
#pragma bit_order [{left | right}]
```

This extension switches the order of bit field assignment.
When **left** is specified, bit field members are assigned from the upper-bit side. When **right** is specified, members are assigned from the lower-bit side.
The default is **right**.
If **left** or **right** is omitted, the order is determined by the option specification.

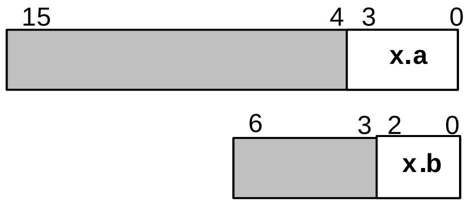
Example

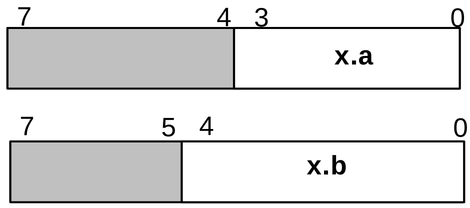
C Source	Bit Assignment

#pragma bit_order right
struct tbl_r {
    unsigned char a:2;
    unsigned char b:3;
} x;
	

#pragma bit_order left
struct tbl_l {
    unsigned char a:2;
    unsigned char b:3;
} y;
	

C Source	Bit Assignment

// Different-size members
#pragma bit_order right
struct tbl_r {
   unsigned short a:4;
   unsigned char b:3;
} x
	

// Larger than the size of the type
#pragma bit_order right
struct tbl_r {
   unsigned char a:4;
   unsigned char b:5;
} x;
	

(8) Alignment Value Specification for Structure Members and Class Members

```
#pragma pack
#pragma unpack
#pragma packoption
```

This changes the alignment values for members of the structure type, union type, and class type declared after the location where **#pragma** is specified in the source program. If neither **#pragma pack** nor **#pragma unpack** is specified or if a structure type, union type, or class type is declared after the location where **#pragma packoption** is specified, the alignment value for members is determined by the **pack** option. Table 4.9 shows the **#pragma** specification and the corresponding alignment values.

Table 4.9 #pragma pack Specifications and Corresponding Member Alignment Values

Member Type	#pragma pack	#pragma unpack	#pragma packoption or No Extension Specification
(signed) char	1	1	1
(unsigned) short	1	2	Determined by the **pack** option
(unsigned) int *, (unsigned) long, (unsigned) long long, floating-point type, and pointer type	1	4	Determined by the **pack** option

Example

```
#pragma pack
struct S1 {
   char a;/* Byte offset = 0*/
   int b;/* Byte offset = 1*/
   char c;/* Byte offset = 5*/
} ST1;/* Total size: 6 bytes*/

#pragma unpack
struct S2 {
   char a;/* Byte offset = 0*/
/* 3-byte empty area*/
   int b;/* Byte offset = 4*/
   char c;/* Byte offset = 8*/
/* 3-byte empty area*/
} ST2;/* Total size: 12 bytes*/
```

The boundary alignment value for structure and class members can also be specified by the **pack** option. When both the option and **#pragma** extension specifier are specified together, the **#pragma** specification takes priority.

(9) Allocation of a Variable to the Absolute Address

```
#pragma address [(]<variable name>=<absolute address>[,...][)]
```

This extension allocates the specified variable to the specified address. The compiler assigns a section for each specified variable, and the variable is allocated to the specified absolute address during linkage. If variables are specified for contiguous addresses, these variables are assigned to a single section.

Example C source description:

```
#pragma address X=0x7f00
int X;
main(){
    X=0;
}
```

Output code:

```
_main:
MOV.L       #0,R5
MOV.L       #7F00H,R14;
MOV.L       R5,[R14]
RTS
.SECTION    $ADDR_B_7F00
.ORG        7F00H
.glb        _X
_X:; static: X
.blkl       1
```

[Remarks]

- Specify **#pragma address** before declaring a variable.
- If an object that is neither a structure/union member nor a variable is specified, an error will be output.
- If **#pragma address** is specified for a single variable more than one time, an error will be output.
- A static variable that is validated by #pragma address and not referred from the source file may be removed by a compiler optimization.
- We recommend not applying #pragma address to a variable which has an initial value but does not have the const qualifier. If this case applies for any variables, take note of the restrictions below.
  - The -rom option (RAMization of the ROM area) of the optimizing linkage editor (rlink) cannot be applied to sections containing such variables.
  - Error messages or warnings will not be displayed for code that includes such a variable.
  - When a section containing such variables is allocated to the RAM, all initial values must be written to the corresponding RAM areas when starting up the program or in advance of that.
- In addition to an "=" sign, a white-space character can also be used as a separator between a variable name and an absolute address.

(10) Endian Specification for Initial Values

```
#pragma endian [{big | little}]
```

This extension specifies the endian for the area that stores static objects.

The specification of this extension is applied from the line containing **#pragma endian** to the end of the file or up to the line immediately before the line containing the next **#pragma endian**.
big specifies big endian. When the **endian=little** option is specified, data is assigned to the section with the section name postfixed with _**B**.
little specifies little endian. When the **endian=big** option is specified, data is assigned to the section with the section name postfixed with _**L**.
When **big** or **little** is omitted, endian is determined by the option specification.

Example When the **endian=little** option is specified (default state)
C source description:

```
#pragma endian big
int A=100;/* D_B section */
#pragma endian
int B=200;/* D section */
```

Output code:

```
   .glb    _A
   .glb    _B
   .SECTION    D,ROMDATA,ALIGN=4
_B:
   .lword   200
   .SECTION   D_B,ROMDATA,ALIGN=4
   .ENDIAN   BIG
_A:
   .lword   100
```

If areas of the **long long** type, **double** type (when the **dbl_size=8** option is specified), and **long double** type (when the **dbl_size=8** option is specified) are included in objects to which **#pragma endian** (differed from the **endian** option) is applied, do not make indirect accesses to these areas using addresses or pointers. In such a case, correct operation will not be guaranteed. If a code that acquires an address in such an area is included, a warning message is displayed.
If bit fields of the **long long** type are included in objects to which **#pragma endian** (differed from the **endian** option) is applied, do not make writes to these areas. In such a case, correct operation will not be guaranteed.
The endian of the following items cannot be changed by this extension. The **endian** option needs to be used.
(1) String literal and initializers for use in the dynamic initialization of aggregates
(2) Branch table of **switch** statement
(3) Objects declared as external references (objects declared through **extern** without initialization expression)
(4) Objects specified as **#pragma address**

(11) Specification of Function in which Instructions at Branch Destinations are Aligned for Execution

```
#pragma instalign4 [(]<function name>[(<branch destination type>)][,...][)]
#pragma instalign8 [(]<function name>[(<branch destination type>)][,...][)]
#pragma noinstalign [(]<function name>[,...][)]
```

Specifies the function in which instructions at branch destinations are aligned for execution.
Instruction allocation addresses in the specified function are adjusted to be aligned to 4-byte boundaries when **#pragma instalign4** is specified or to 8-byte boundaries when **#pragma instalign8** is specified.
In the function specified with **#pragma noinstalign**, alignment of allocation addresses is not adjusted.
The branch destination type should be selected from the following*:
No specification:Head of function and **case** and **default** labels of **switch** statement
inmostloop: Head of each inmost loop, head of function, and **case** and **default** labels of **switch** statement
loop: Head of each loop, head of function, and **case** and **default** labels of **switch** statement

Note Alignment is adjusted only for the branch destinations listed above; alignment of the other destinations is not adjusted. For example, when **loop** is selected, alignment of the head of a loop is adjusted but alignment is not adjusted at the branch destination of an **if** statement that is used in the loop but does not generate a loop.

Except that each **#pragma** extension specification is valid only in the specified function, these specifiers work in the same way as the **instalign4**, **instalign8**, and **noinstalign** options. When both the options and **#pragma** extension specifiers are specified together, the **#pragma** specifications take priority.
In the code section that contains a function specified with **instalign4** or **instalign8**, the alignment value is changed to 4 (**instalign4** is specified) or 8 (**instalign8** is specified). If a single code section contains both a function specified with **instalign4** and that specified with **instalign8**, the alignment value in the code section is set to 8.
The other detailed functions of these **#pragma** extension specifiers are the same as those of the **instalign4**, **instalign8**, and **noinstalign** options; refer to the description of each option.

(12) Specification of Function for generating a code for detection of stack smashing [Professional Edition only] [V2.04.00 or later]
#pragma stack_protector [(] function name [(num=<integer value>)] [,...] [)]
#pragma no_stack_protector [(] function name [,...] [)]

Generates a code for detection of stack smashing at the entry and the end of a function. A code for detection of stack smashing consists of instructions executing the three processes shown below.
(1) A 4-byte area is allocated just before (in the direction towards address 0xFFFFFFFF) the local variable area at the entry to a function, and the value specified by <number> is stored in the allocated area.
(2) At the end of the function, whether the 4-byte area in which <number> was stored has been rewritten is checked.
(3) If the 4-byte area has been rewritten in (2), the **__stack_chk_fail** function is called as the stack has been smashed.
A decimal number from 0 to 4294967295 should be specified in <number>. If the specification of <number> is omitted, the compiler automatically selects the number.
The **__stack_chk_fail** function needs to be defined by the user. It should contain postprocesses for the detected stack smashing.
Note the following items when defining the **__stack_chk_fail** function.

- The only possible type of return value is void and any formal parameters not allowed.
- It is prohibited to call the **__stack_chk_fail** function as a normal function.
- The **__stack_chk_fail** function does not generate a code for detection of stack smashing regardless of the **-stack_protector** and **-stack_protector_all** options, and **#pragma stack_protector**.
- In a C++ program, add extern "C" to the definition or the declaration for **__stack_chk_fail** function.
- Prevent returning to the caller (the function where stack smashing was detected) by taking measures such as calling **abort()** in **__stack_chk_fail** function and terminating the program.
- Do not define the function as **static**.

A code for detection of stack smashing is not generated for a function for which **#pragma no_stack_protector** has been specified regardless of the **-stack_protector** option and **-stack_protector_all** option.
If these options are used simultaneously with **#pragma stack_protector**, the **-stack_protector** option, or the **-stack_protector_all** option, the specification by **#pragma** becomes valid.
An error will occur when **#pragma stack_protector** and **#pragma no_stack_protector** are specified simultaneously for the same function within a single translation unit.
When the function specified by **#pragma stack_protector** is specified as any one of the following functions, an error message is output.
#pragma inline
#pragma inline_asm
#pragma entry

4.2.5 Using a Keyword

This section explains using the following keyword.

- Description of access in specified size

(1) Description of Access in Specified Size

```
__evenaccess <type specifier> <variable name>
<type specifier> __evenaccess <variable name>
```

A variable is accessed in the declared or defined size.
This extension guarantees access in the size of the target variable.
This keyword is applicable to 4-byte or smaller scalar types.

Example C source description:

```
#pragma address A=0xff0178
unsigned long __evenaccess A;
void test(void)
{
    A &= ~0x20;
}
```

Output code (__**evenaccess** not specified):

```
_test:
 MOV.L #16712056,R1
 BCLR #5,[R1]          ; Memory access in 1 byte
 RTS
```

Output code (__**evenaccess** specified):

```
_test:
 MOV.L #16712056,R1
 MOV.L [R1],R5         ; Memory access in 4 bytes
 BCLR  #5,R5
 MOV.L R5,[R1]         ; Memory access in 4 bytes
 RTS
```

The __**evenaccess** is invalid to the case of accessing of members by a lump of these structure and union frame. When __**evenaccess** is specified for a structure or a union, __**evenaccess** is applied to all members. In this case, the access size is guaranteed for 4-byte or smaller scalar integer types, but the size of access in structure or union units is not guaranteed.

4.2.6 Intrinsic Functions

In the CCRX, some of the assembler instructions can be described in C source as "Intrinsic Functions". However, it is not described "as assembler instruction", but as a function format set in the CCRX. When an intrinsic function is used, the compiler inserts the corresponding code into the program.

Table 4.10 Intrinsic Functions

No.	Item	Specifications	Function	Restriction in User Mode*
1	Maximum value and minimum value	signed long max(signed long data1, signed long data2)	Selects the maximum value.	O
		signed long __max(signed long data1, signed long data2) [V2.05.00 or later]		
2		signed long min(signed long data1, signed long data2)	Selects the minimum value.	O
		signed long __min(signed long data1, signed long data2) [V2.05.00 or later]		
3	Byte switch	unsigned long revl(unsigned long data)	Reverses the byte order in longword data.	O
		unsigned long __revl(unsigned long data) [V2.05.00 or later]		
4		unsigned long revw(unsigned long data)	Reverses the byte order in longword data in word units.	O
		unsigned long __revw(unsigned long data) [V2.05.00 or later]		
5	Data exchange	void xchg(signed long *data1, signed long *data2)	Exchanges data.	O
		void __xchg(signed long *data1, signed long *data2) [V2.05.00 or later]		
6	Multi-ply-and-accu-mulate operation	long long rmpab(long long init, unsigned long count, signed char *addr1, signed char *addr2)	Multiply-and-accumulate operation (byte).	O
		long long __rmpab(long long init, unsigned long count, signed char *addr1, signed char *addr2) [V2.05.00 or later]		
7		long long rmpaw(long long init, unsigned long count, short *addr1, short *addr2)	Multiply-and-accumulate operation (word).	O
		long long __rmpaw(long long init, unsigned long count, short *addr1, short *addr2) [V2.05.00 or later]		
8		long long rmpal(long long init, unsigned long count, long *addr1, long *addr2)	Multiply-and-accumulate operation (longword).	O
		long long __rmpal(long long init, unsigned long count, long *addr1, long *addr2) [V2.05.00 or later]		

No.	Item	Specifications	Function	Restriction in User Mode*
9	Rotation	unsigned long rolc(unsigned long data)	Rotates data including the carry to left by one bit.	O
		unsigned long __rolc(unsigned long data) [V2.05.00 or later]		
10		unsigned long rorc(unsigned long data)	Rotates data including the carry to right by one bit.	O
		unsigned long __rorc(unsigned long data) [V2.05.00 or later]		
11		unsigned long rotl(unsigned long data, unsigned long num)	Rotates data to left.	O
		unsigned long __rotl(unsigned long data, unsigned long num) [V2.05.00 or later]		
12		unsigned long rotr(unsigned long data, unsigned long num)	Rotates data to right.	O
		unsigned long __rotr(unsigned long data, unsigned long num) [V2.05.00 or later]		
13	Special instructions	void brk(void)	**BRK** instruction exception.	O
		void __brk(void) [V2.05.00 or later]		
14		void int_exception(signed long num)	**INT** instruction exception.	O
		void __int_exception(signed long num) [V2.05.00 or later]		
15		void wait(void)	Stops program execution.	×
		void __wait(void) [V2.05.00 or later]		
16		void nop(void)	Expanded to a **NOP** instruction.	O
		void __nop(void) [V2.05.00 or later]		
17	Processor interrupt priority level (IPL)	void set_ipl(signed long level)	Sets the interrupt priority level.	×
		void __set_ipl(signed long level) [V2.05.00 or later]		
18		unsigned char get_ipl(void)	Refers to the interrupt priority level.	O
		unsigned char __get_ipl(void) [V2.05.00 or later]		
19	Processor status word (PSW)	void set_psw(unsigned long data)	Sets a value for **PSW**.	Δ
		void __set_psw(unsigned long data) [V2.05.00 or later]		
20		unsigned long get_psw(void)	Refers to **PSW** value.	O
		unsigned long __get_psw(void) [V2.05.00 or later]		

No.	Item	Specifications	Function	Restriction in User Mode*
21	Floating-point status word (FPSW)	void set_fpsw(unsigned long data)	Sets a value for **FPSW**.	O
		void __set_fpsw(unsigned long data) [V2.05.00 or later]		
22		unsigned long get_fpsw(void)	Refers to **FPSW** value.	O
		unsigned long __get_fpsw(void) [V2.05.00 or later]		
23	User stack pointer (USP)	void set_usp(void *data)	Sets a value for **USP**.	O
		void __set_usp(void *data) [V2.05.00 or later]		
24		void *get_usp(void)	Refers to **USP** value.	O
		void *__get_usp(void) [V2.05.00 or later]		
25	Interrupt stack pointer (ISP)	void set_isp(void *data)	Sets a value for **ISP**.	Δ
		void __set_isp(void *data) [V2.05.00 or later]		
26		void *get_isp(void)	Refers to **ISP** value.	O
		void *__get_isp(void) [V2.05.00 or later]		
27	Interrupt table register (INTB)	void set_intb(void *data)	Sets a value for **INTB**.	Δ
		void __set_intb(void *data) [V2.05.00 or later]		
28		void *get_intb(void)	Refers to **INTB** value.	O
		void *__get_intb(void) [V2.05.00 or later]		
29	Backup PSW (BPSW)	void set_bpsw(unsigned long data)	Sets a value for **BPSW**.	Δ
		void __set_bpsw(unsigned long data) [V2.05.00 or later]		
30		unsigned long get_bpsw(void)	Refers to **BPSW** value.	O
		unsigned long __get_bpsw(void) [V2.05.00 or later]		
31	Backup PC (BPC)	void set_bpc(void *data)	Sets a value for **BPC**.	Δ
		void __set_bpc(void *data) [V2.05.00 or later]		
32		void *get_bpc(void)	Refers to **BPC** value.	O
		void *__get_bpc(void) [V2.05.00 or later]		
33	Fast interrupt vector register (FINTV)	void set_fintv(void *data)	Sets a value for **FINTV**.	Δ
		void __set_fintv(void *data) [V2.05.00 or later]		
34		void *get_fintv(void)	Refers to **FINTV** value.	O
		void *__get_fintv(void) [V2.05.00 or later]		

No.	Item	Specifications	Function	Restriction in User Mode*
35	Significant 64-bit multipli-cation	signed long long emul(signed long data1, signed long data2)	Signed multiplication of significant 64 bits.	O
		signed long long __emul(signed long data1, signed long data2) [V2.05.00 or later]		
36		unsigned long long emulu(unsigned long data1, unsigned long data2)	Unsigned multiplication of significant 64 bits.	O
		unsigned long long __emulu(unsigned long data1, unsigned long data2) [V2.05.00 or later]		
37	Processor mode (**PM**)	void chg_pmusr(void)	Switches to user mode.	Δ
		void __chg_pmusr(void) [V2.05.00 or later]		
38	Accumulator (**ACC**)	void set_acc(signed long long data)	Sets the **ACC**.	O
		void __set_acc(signed long long data) [V2.05.00 or later]		
39		signed long long get_acc(void)	Refers to the **ACC**.	O
		signed long long __get_acc(void) [V2.05.00 or later]		
40	Control of the interrupt enable bits	void setpsw_i(void)	Sets the interrupt enable bit to 1.	Δ
		void __setpsw_i(void) [V2.05.00 or later]		
41		void clrpsw_i(void)	Clears the interrupt enable bit to 0.	Δ
		void __clrpsw_i(void) [V2.05.00 or later]		
42	Multi-ply-and-accu-mulate operation	long macl(short *data1, short *data2, unsigned long count)	Multiply-and-accumulate operation of 2-byte data.	O
		long __macl(short *data1, short *data2, unsigned long count) [V2.05.00 or later]		
43		short macw1(short *data1, short *data2, unsigned long count) short macw2(short *data1, short *data2, unsigned long count)	Multiply-and-accumulate operation of fixed-point data.	O
		short __macw1(short *data1, short *data2, unsigned long count) [V2.05.00 or later] short __macw2(short *data1, short *data2, unsigned long count) [V2.05.00 or later]		
44	Exception vec-tor table register (EXTB)	void set_extb(void *data)	Sets a value for **EXTB**.	Δ
		void __set_extb(void *data) [V2.05.00 or later]		
45		void *get_extb(void)	Refers to **EXTB** value.	O
		void *__get_extb(void) [V2.05.00 or later]		

No.	Item	Specifications	Function	Restriction in User Mode*
46	Bit manipulation	void __bclr(unsigned char *data, unsigned long bit) [V2.05.00 or later]	Clears one bit.	O
47		void __bset(unsigned char *data, unsigned long bit) [V2.05.00 or later]	Sets one bit.	O
48		void __bnot(unsigned char *data, unsigned long bit) [V2.05.00 or later]	Reverses one bit.	O
49	Double-precision floating-point status word (DPSW)	void __set_dpsw(unsigned long data) [V3.01.00 or later]	Sets a value in DPSW.	O
50		unsigned long __get_dpsw(void) [V3.01.00 or later]	Refers to the DPSW value.	O
51	Double-precision floating-point exception handling operation control register (DECNT)	void __set_decnt(unsigned long data) [V3.01.00 or later]	Sets a value in DECNT.	O
52		unsigned long __get_decnt(void) [V3.01.00 or later]	Refers to the DECNT value.	O
53	Double-precision floating-point exception program counter (DEPC)	void *__get_depc(void) [V3.01.00 or later]	Refers to the DEPC value.	O

No.	Item	Specifications	Function	Restriction in User Mode*
54	Trigonometric function unit	void __init_tfu(void) [V3.01.00 or later]	Initializes the trigonometric function unit.	O
55		void __sincosf(float f, float *s, float *c) [V3.01.00 or later]	Uses the trigonometric function unit to calculate the sine and cosine of an angle at the same time (single precision).	O
56		void __atan2hypotf(float y, float x, float *a, float *h) [V3.01.00 or later]	Uses the trigonometric function unit to calculate the arc tangent of x and y and the square root of the sum of squares of these values ($\sqrt{x^2+y^2}$) at the same time (single precision).	O
57		void __sincosfx(signed long fx, signed long *s, signed long *c) [V3.05.00 or later]	Uses the trigonometric function unit to calculate the sine and cosine of an angle at the same time (fixed-point number).	O
58		signed long __sinfx(signed long fx) [V3.05.00 or later]	Uses the trigonometric function unit to calculate the sine of an angle (fixed-point number).	O
59		signed long __cosfx(signed long fx) [V3.05.00 or later]	Uses the trigonometric function unit to calculate the cosine of an angle (fixed-point number).	O
60		void __atan2hypotfx(signed long y, signed long x, signed long *a, signed long *h) [V3.05.00 or later]	Uses the trigonometric function unit to calculate the arc tangent of x and y and the square root of the sum of squares of these values ($\sqrt{x^2+y^2}$) at the same time (fixed-point number).	O
61		signed long __atan2fx(signed long y, signed long x) [V3.05.00 or later]	Uses the trigonometric function unit to calculate the arc tangent of x and y (fixed-point number).	O
62		signed long __hypotfx(signed long x, signed long y) [V3.05.00 or later]	Uses the trigonometric function unit to calculate the square root of the sum of squares of x and y ($\sqrt{x^2+y^2}$) (fixed-point number).	O

Note * Indicates whether the function is limited when the RX processor mode is user mode.
O: Has no restriction.
×: Must not be used in user mode because a privileged instruction exception occurs.
Δ: Has no effect when executed in user mode.

signed long max(signed long data1, signed long data2)
signed long __max(signed long data1, signed long data2) [V2.05.00 or later]

[Description]
Selects the greater of two input values (this function is expanded into a **MAX** instruction).
[Header]
<machine.h>
[Parameters]
data1 Input value 1
data2 Input value 2
[Return value]
The greater value of **data1** and **data2**
[Example]

```
#include <machine.h>
extern signed long ret,in1,in2;
void main(void)
{
    ret = max(in1,in2);// Stores the greater value of in1 and in2 in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

signed long min(signed long data1, signed long data2)
signed long __min(signed long data1, signed long data2) [V2.05.00 or later]

[Description]
Selects the smaller of two input values (this function is expanded into a MIN instruction).
[Header]
<machine.h>
[Parameters]
data1 Input value 1
data2 Input value 2
[Return value]
The smaller value of data1 and data2
[Example]

```
#include <machine.h>
extern signed long ret,in1,in2;
void main(void)
{
    ret = min(in1,in2);// Stores the smaller value of in1 and in2 in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

unsigned long revl(unsigned long data)
unsigned long __revl(unsigned long data) [V2.05.00 or later]

[Description]
Reverses the byte order in 4-byte data (this function is expanded into a REVL instruction).
[Header]
<machine.h>
[Parameters]
data Data for which byte order is to be reversed
[Return value]
Value of data with the byte order reversed
[Example]

```
#include <machine.h>
extern unsigned long ret,indata=0x12345678;
void main(void)
{
    ret = revl(indata);// ret = 0x78563412
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

unsigned long revw(unsigned long data)
unsigned long __revw(unsigned long data) [V2.05.00 or later]

[Description]
Reverses the byte order within each of the upper and lower two bytes of 4-byte data (this function is expanded into a REVW instruction).
[Header]
<machine.h>
[Parameters]
data Data for which byte order is to be reversed
[Return value]
Value of data with the byte order reversed within the upper and lower two bytes
[Example]

```
#include <machine.h>
extern unsigned long ret;indata=0x12345678;
void main(void)
{
    ret = revw(indata);// ret = 0x34127856
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

void xchg(signed long *data1, signed long *data2)
void __xchg(signed long *data1, signed long *data2) [V2.05.00 or later]

[Description]
Exchanges the contents of the areas indicated by parameters (this function is expanded into an XCHG instruction).
[Header]
<machine.h>
[Parameters]
*data1 Input value 1
*data2 Input value 2
[Example]

```
#include <machine.h>
extern signed long *in1,*in2;
void main(void)
{
    xchg (in1,in2);// Exchanges data at address in1 and address in2.
}
```

[Remarks]
The XCHG instruction to be generated has a memory operand with a location indicated by **data2**.
The header does not have to be included when using an intrinsic function whose name starts with __.

long long rmpab(long long init, unsigned long count, signed char *addr1, signed char *addr2)
long long __rmpab(long long init, unsigned long count, signed char *addr1, signed char *addr2) [V2.05.00 or later]

[Description]
Performs a multiply-and-accumulate operation with the initial value specified by init, the number of multiply-and-accumulate operations specified by count, and the start addresses of values to be multiplied specified by addr1 and addr2 (this function is expanded into a RMPA.B instruction).
[Header]
<machine.h>
[Parameters]
init Initial value
count Count of multiply-and-accumulate operations
*addr1 Start address of values 1 to be multiplied
*addr2 Start address of values 2 to be multiplied
[Return value]
Lower 64 bits of the init + Σ(data1[n] * data2[n]) result (n = 0, 1, ..., const - 1)
[Example]

```
#include <machine.h>
extern signed char data1[8],data2[8];
long long sum;
void main(void)
{
    sum=rmpab(0, 8, data1, data2);
        // Specifies 0 as the initial value, adds the result
        // of multiplication of arrays data1 and data2,
        // and stores the result in sum.
}
```

[Remark]
The **RMPA** instruction obtains a result in a maximum of 80 bits, but this intrinsic function handles only 64 bits.
The header does not have to be included when using an intrinsic function whose name starts with __.

long long rmpaw(long long init, unsigned long count, short *addr1, short *addr2) long long __rmpaw(long long init, unsigned long count, short *addr1, short *addr2) [V2.05.00 or later]

[Description]

Performs a multiply-and-accumulate operation with the initial value specified by init, the number of multiply-and-accumulate operations specified by count, and the start addresses of values to be multiplied specified by addr1 and addr2 (this function is expanded into a RMPA.W instruction).

[Header]

<machine.h>

[Parameters]

init   Initial value

count   Count of multiply-and-accumulate operations

*addr1   Start address of values 1 to be multiplied

*addr2   Start address of values 2 to be multiplied

[Return value]

Lower 64 bits of the init + Σ(data1[n] * data2[n]) result (n = 0, 1, ..., const - 1)

[Example]

```
#include <machine.h>
extern signed short data1[8],data2[8];
long long sum;
void main(void)
{
    sum=rmpaw(0, 8, data1, data2);
        // Specifies 0 as the initial value, adds the result
        // of multiplication of arrays data1 and data2,
        // and stores the result in sum.
}
```

[Remark]

The **RMPA** instruction obtains a result in a maximum of 80 bits, but this intrinsic function handles only 64 bits.
The header does not have to be included when using an intrinsic function whose name starts with __.

```
long long rmpal(long long init, unsigned long count, long *addr1, long *addr2)
long long __rmpal(long long init, unsigned long count, long *addr1, long *addr2) [V2.05.00 or later]
```

[Description]

Performs a multiply-and-accumulate operation with the initial value specified by init, the number of multiply-and-accumulate operations specified by count, and the start addresses of values to be multiplied specified by addr1 and addr2 (this function is expanded into a RMPA.L instruction).

[Header]

<machine.h>

[Parameters]

init Initial value

count Count of multiply-and-accumulate operations

*addr1 Start address of values 1 to be multiplied

*addr2 Start address of values 2 to be multiplied

[Return value]

Lower 64 bits of the init + Σ(data1[n] * data2[n]) result (n = 0, 1, ..., const - 1)

[Example]

```
#include <machine.h>
extern signed long data1[8],data2[8];
long long sum;
void main(void)
{
    sum=rmpal(0, 8, data1, data2);
        // Specifies 0 as the initial value, adds the result
        // of multiplication of arrays data1 and data2,
        // and stores the result in sum.
}
```

[Remarks]

The **RMPA** instruction obtains a result in a maximum of 80 bits, but this intrinsic function handles only 64 bits.

The header does not have to be included when using an intrinsic function whose name starts with __.

unsigned long rolc(unsigned long data) unsigned long __rolc(unsigned long data) [V2.05.00 or later]

[Description]
Rotates data including the C flag to left by one bit (this function is expanded into a ROLC instruction).
The bit pushed out of the operand is set to the C flag.

[Header]
<machine.h>

[Parameters]
data Data to be rotated to left

[Return value]
Result of 1-bit left rotation of data including the C flag

[Example]

```
#include <machine.h>
extern unsigned long ret, indata;
void main(void)
{
    ret = rolc(indata);// Rotates indata including the C flag
        // to left by one bit and stores the result
        // in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

unsigned long rorc(unsigned long data) unsigned long __rorc(unsigned long data) [V2.05.00 or later]

[Description]
Rotates data including the C flag to right by one bit (this function is expanded into a RORC instruction).
The bit pushed out of the operand is set to the C flag.

[Header]
<machine.h>

[Parameters]
data Data to be rotated to right

[Return value]
Result of 1-bit right rotation of data including the C flag

[Example]

```
#include <machine.h>
extern unsigned long ret, indata;
void main(void)
{
    ret = rorc(indata);// Rotates indata including the C flag
        // to right by one bit and stores the result
        // in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

```
unsigned long rotl(unsigned long data, unsigned long num)
unsigned long __rotl(unsigned long data, unsigned long num) [V2.05.00 or later]
```

[Description]
Rotates data to left by the specified number of bits (this function is expanded into a ROTL instruction).
The bit pushed out of the operand is set to the C flag.

[Header]
<machine.h>

[Parameters]

data	Data to be rotated to left
num	Number of bits to be rotated

[Return value]
Result of num-bit left rotation of data

[Example]

```
#include <machine.h>
extern unsigned long ret, indata;
void main(void)
{
    ret = rotl(indata, 31); // Rotates indata to left by 31 bits
    // and stores the result in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

```
unsigned long rotr(unsigned long data, unsigned long num)
unsigned long __rotr(unsigned long data, unsigned long num) [V2.05.00 or later]
```

[Description]
Rotates data to right by the specified number of bits (this function is expanded into a ROTR instruction).
The bit pushed out of the operand is set to the C flag.

[Header]
<machine.h>

[Parameters]

data	Data to be rotated to right
num	Number of bits to be rotated

[Return value]
Result of num-bit right rotation of data

[Example]

```
#include <machine.h>
extern unsigned long ret, indata;
void main(void)
{
    ret = rotr(indata, 31); // Rotates indata to right by 31 bits
        // and stores the result in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

void brk(void)
void __brk(void) [V2.05.00 or later]

[Description]
This function is expanded into a BRK instruction.
[Header]
<machine.h>
[Parameters]
-
[Return value]
-
[Example]

```
#include <machine.h>
void main(void)
{
    brk();// BRK instruction
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

void int_exception(signed long num)
void __int_exception(signed long num) [V2.05.00 or later]

[Description]
This function is expanded into an INT num instruction.
[Header]
<machine.h>
[Parameters]
num INT instruction number
[Return value]
-
[Example]

```
#include <machine.h>
void main(void)
{
    int_exception(10);// INT #10 instruction
}
```

[Remarks]
Only an integer from 0 to 255 can be specified as **num**.
The header does not have to be included when using an intrinsic function whose name starts with __.

void wait(void)
void __wait(void) [V2.05.00 or later]

[Description]
This function is expanded into a WAIT instruction.
[Header]
<machine.h>
[Parameters]
-
[Return value]
-
[Example]

```
#include <machine.h>
void main(void)
{
    wait();// WAIT instruction
}
```

[Remarks]
This function must not be executed when the RX processor mode is user mode. If executed, a privileged instruction exception of the RX occurs due to the specifications of the **WAIT** instruction.
The header does not have to be included when using an intrinsic function whose name starts with __.

void nop(void)
void __nop(void) [V2.05.00 or later]

[Description]
This function is expanded into a NOP instruction.
[Header]
<machine.h>
[Parameters]
-
[Return value]
-
[Example]

```
#include <machine.h>
void main(void)
{
    nop();// NOP instruction
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

void set_ipl(signed long level)
void __set_ipl(signed long level) [V2.05.00 or later]

[Description]
Changes the interrupt mask level.
[Header]
<machine.h>
[Parameters]
-
[Return value]
level Interrupt mask level to be set
[Example]

```
#include <machine.h>
void main(void)
{
    set_ipl(7);// Sets PSW.IPL to 7.
}
```

[Remarks]
A value from 0 to 15 can be specified for level by default, and a value from 0 to 7 can be specified when -patch=rx610 is specified.
If a value outside the above range is specified when level is a constant, an error will be output.
This function must not be executed when the RX processor mode is user mode. If executed, a privileged instruction exception of the RX occurs due to the specifications of the MVTIPL instruction.
The header does not have to be included when using an intrinsic function whose name starts with __.

unsigned char get_ipl(void)
unsigned char __get_ipl(void) [V2.05.00 or later]

[Description]
Refers to the interrupt mask level.
[Header]
<machine.h>
[Parameters]
-
[Return value]
Interrupt mask level
[Example]

```
#include <machine.h>
extern unsigned char level;
void main(void)
{
    level=get_ipl();// Obtains the PSW.IPL value and
        // stores it in level.
}
```

[Remarks]
If a value smaller than 0 or greater than 7 is specified as level, an error will be output.
The header does not have to be included when using an intrinsic function whose name starts with __.

void set_psw(unsigned long data)
void __set_psw(unsigned long data) [V2.05.00 or later]

[Description]
Sets a value to PSW.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
    set_psw(data);// Sets PSW to the value specified by data.
}
```

[Remarks]
Due to the specifications of the RX instruction set, a write to the **PM** bit of **PSW** is ignored. In addition, a write to **PSW** is ignored when the RX processor mode is user mode.
The header does not have to be included when using an intrinsic function whose name starts with __.

unsigned long get_psw(void)
unsigned long __get_psw(void) [V2.05.00 or later]

[Description]
Refers to the PSW value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
PSW value
[Example]

```
#include <machine.h>
extern unsigned long ret;
void main(void)
{
    ret=get_psw();// Obtains the PSW value and stores it in ret.
}
```

[Remarks]
In some cases, the timing at which the **PSW** value is obtained differs from the timing at which **get_psw** was called, due to the effect of optimization. Therefore when a code using the **C**, **Z**, **S**, or **O** flag included in the return value of this function is written after some sort of operation, correct operation will not be guaranteed.
The header does not have to be included when using an intrinsic function whose name starts with __.

void set_fpsw(unsigned long data)
void __set_fpsw(unsigned long data) [V2.05.00 or later]

[Description]
Sets a value to FPSW.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
    set_fpsw(data);// Sets FPSW to the value specified by data.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

unsigned long get_fpsw(void)
unsigned long __get_fpsw(void) [V2.05.00 or later]

[Description]
Refers to the FPSW value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
FPSW value
[Example]

```
#include <machine.h>
extern unsigned long ret;
void main(void)
{
    ret=get_fpsw();// Obtains the FPSW value and stores it
        // in ret.
}
```

[Remarks]
In some cases, the timing at which the **FPSW** value is obtained differs from the timing at which **get_fpsw** was called, due to the effect of optimization. Therefore when a code using the **CV**, **CO**, **CZ**, **CU**, **CX**, **CE**, **FV**, **FO**, **FZ**, **FU**, **FX**, or **FS** flag included in the return value of this function is written after some sort of operation, correct operation will not be guaranteed.
The header does not have to be included when using an intrinsic function whose name starts with __.

void set_usp(void *data)
void __set_usp(void *data) [V2.05.00 or later]

[Description]
Sets a value to USP.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
#include <machine.h>
extern void * data;
void main(void)
{
    set_usp(data);// Sets USP to the value specified by data.
}
```

[Remarks]
A 4-byte boundary address should be specified as **data**.
Program operation is not guaranteed when a 1-byte boundary address or 2-byte boundary address is specified.
The header does not have to be included when using an intrinsic function whose name starts with __.

void *get_usp(void)
void *__get_usp(void) [V2.05.00 or later]

[Description]
Refers to the USP value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
USP value
[Example]

```
#include <machine.h>
extern void * ret;
void main(void)
{
    ret=get_usp();// Obtains the USP value and stores it in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

```
void set_isp(void *data)
void __set_isp(void *data) [V2.05.00 or later]
```

[Description]
Sets a value to ISP.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
#include <machine.h>
extern void * data;
void main(void)
{
    set_isp(data);// Sets ISP to the value specified by data.
}
```

[Remarks]
Due to the specifications of the **MVTC** instruction used in this function, a write to **ISP** is ignored when the RX processor mode is user mode.
A 4-byte boundary address should be specified as **data**.
Program operation is not guaranteed when a 1-byte boundary address or 2-byte boundary address is specified.
The header does not have to be included when using an intrinsic function whose name starts with __.

```
void *get_isp(void)
void *__get_isp(void) [V2.05.00 or later]
```

[Description]
Refers to the ISP value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
ISP value
[Example]

```
#include <machine.h>
extern void * ret;
void main(void)
{
    ret=get_isp();// Obtains the ISP value and stores it in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

void set_intb (void *data)
void __set_intb(void *data) [V2.05.00 or later]

[Description]
Sets a value to INTB.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
#include <machine.h>
extern void * data;
void main(void)
{
    set_intb (data);// Sets INTB to the value specified by data.
}
```

[Remarks]
Due to the specifications of the **MVTC** instruction used in this function, a write to **INTB** is ignored when the RX processor mode is user mode.
The header does not have to be included when using an intrinsic function whose name starts with __.

void *get_intb(void)
void *__get_intb(void) [V2.05.00 or later]

[Description]
Refers to the INTB value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
INTB value
[Example]

```
#include <machine.h>
extern void * ret;
void main(void)
{
    ret=get_intb();// Obtains the INTB value and stores it in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

void set_bpsw(unsigned long data) void __set_bpsw(unsigned long data) [V2.05.00 or later]

[Description]
Sets a value to BPSW.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
    set_bpsw (data);// Sets BPSW to the value specified by data.
}
```

[Remarks]
Due to the specifications of the **MVTC** instruction used in this function, a write to **BPSW** is ignored when the RX processor mode is user mode.
The header does not have to be included when using an intrinsic function whose name starts with __.

unsigned long get_bpsw(void) unsigned long __get_bpsw(void) [V2.05.00 or later]

[Description]
Refers to the BPSW value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
BPSW value
[Example]

```
#include <machine.h>
extern unsigned long ret;
void main(void)
{
    ret=get_bpsw ();// Obtains the BPSW value and stores it
        // in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

void set_bpc(void *data)
void __set_bpc(void *data) [V2.05.00 or later]

[Description]
Sets a value to BPC.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
#include <machine.h>
extern void * data;
void main(void)
{
    set_bpc(data);// Sets BPC to the value specified by data.
}
```

[Remarks]
Due to the specifications of the **MVTC** instruction used in this function, a write to **BPC** is ignored when the RX processor mode is user mode.
The header does not have to be included when using an intrinsic function whose name starts with __.

void *get_bpc(void)
void *__get_bpc(void) [V2.05.00 or later]

[Description]
Refers to the BPC value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
BPC value
[Example]

```
#include <machine.h>
extern void * ret;
void main(void)
{
    ret=get_bpc();// Obtains the BPC value and stores it in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

void set_fintv(void *data)
void __set_fintv(void *data) [V2.05.00 or later]

[Description]
Sets a value to FINTV.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]

[Example]

```
#include <machine.h>
extern void * data;
void main(void)
{
    set_fintv(data);// Sets FINTV to the value specified by data.
}
```

[Remarks]
Due to the specifications of the **MVTC** instruction used in this function, a write to **FINTV** is ignored when the RX processor mode is user mode.
The header does not have to be included when using an intrinsic function whose name starts with __.

void *get_fintv(void)
void *__get_fintv(void) [V2.05.00 or later]

[Description]
Refers to the FINTV value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
FINTV value
[Example]

```
#include <machine.h>
extern void * ret;
void main(void)
{
    ret=get_fintv();// Obtains the FINTV value and stores it
        // in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

signed long long emul(signed long data1, signed long data2) signed long long __emul(signed long data1, signed long data2) [V2.05.00 or later]

[Description]
Performs signed multiplication of significant 64 bits.
[Header]
<machine.h>
[Parameters]
data 1 Input value 1
data 2 Input value 2
[Return value]
Result of signed multiplication (signed 64-bit value)
[Example]

```
#include <machine.h>
extern signed long long ret;
extern signed long data1, data2;
void main(void)
{
    ret=emul(data1, data2);// Calculates the value of
        // "data1 * data2" and stores it in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

unsigned long long emulu(unsigned long data1, unsigned long data2) unsigned long long __emulu(unsigned long data1, unsigned long data2) [V2.05.00 or later]

[Description]
Performs unsigned multiplication of significant 64 bits.
[Header]
<machine.h>
[Parameters]
data 1 Input value 1
data 2 Input value 2
[Return value]
Result of unsigned multiplication (unsigned 64-bit value)
[Example]

```
#include <machine.h>
extern unsigned long long ret;
extern unsigned long data1, data2;
void main(void)
{
    ret=emulu(data1, data2);// Calculates the value of
        // "data1 * data2" and stores it in ret.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

void chg_pmusr(void)
void __chg_pmusr(void) [V2.05.00 or later]

[Description]
Switches the RX processor mode to user mode.
[Header]
<machine.h>
[Parameters]
-
[Return value]
-
[Example]

```
#include <machine.h>
void main(void);
void Do_Main_on_UserMode(void)
{
    chg_pmusr();// Switches the RX processor mode to user mode.
    main();// Executes the main function.
}
```

[Remarks]
This function is provided for a reset processing function or interrupt function. Usage in any other function is not recommended.
The processor mode is not switched when the RX processor mode is user mode.
Since the stack is switched from the interrupt stack to the user stack when this function is executed, the following conditions must be met in a function that is calling this function. If the conditions are not met, code does not operate correctly because the stack is not the same before and after this function has been executed.

- Execution cannot be returned to the calling function.
- The **auto** variable cannot be declared.
- Parameters cannot be declared.

The header does not have to be included when using an intrinsic function whose name starts with __.

void set_acc(signed long long data)
void __set_acc(signed long long data) [V2.05.00 or later]

[Description]
Sets a value to ACC.
[Header]
<machine.h>
[Parameters]
data Value to be set to ACC
[Return value]
-
[Example]

```
#include <machine.h>
void main(void)
{
    signed long long data = 0x123456789ab0000LL;
    set_acc(data);// Sets ACC to the value specified by data.
}
```

[Remarks]
The header does not have to be included when using an intrinsic function whose name starts with __.

signed long long get_acc(void)
signed long long __get_acc(void) [V2.05.00 or later]

[Description]
Refers to the ACC value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
ACC value
[Example]

```
/* Example of program using the function to save and restore ACC by*/
/* get_acc and set_acc*/
#include <machine.h>
signed long a, b, c;
void func(void)
{
    signed long long bak_acc = get_acc();
        // Obtains the ACC value and saves it
        // in bak_acc.
    c = a * b;// Multiplication (ACC is damaged).
    set_acc(bak_acc);// Restores ACC with a value saved by
        // bak_acc.
}
```

[Remarks]
Due to the specifications of the RX instruction set, contents in the lower 16 bits of **ACC** cannot be obtained. This function returns the value of 0 for these bits.
The header does not have to be included when using an intrinsic function whose name starts with __.

void setpsw_i(void)
void __setpsw_i(void) [V2.05.00 or later]

[Description]
Sets the interrupt enable bit (I bit) in PSW to 1.
[Header]
<machine.h>
[Parameters]
-
[Return value]
-
[Example]

```
#include <machine.h>
void main(void)
{
    setpsw_i();// Sets the interrupt enable bit to 1.
}
```

[Remarks]
Due to the specifications of the SETPSW instruction used by this function, writing to the interrupt enable bit is ignored when the RX processor mode is set to user mode.
The header does not have to be included when using an intrinsic function whose name starts with __.

void clrpsw_i(void)
void __clrpsw_i(void) [V2.05.00 or later]

[Description]
Clears the interrupt enable bit (I bit) in PSW to 0.
[Header]
<machine.h>
[Parameters]
-
[Return value]
-
[Example]

```
#include <machine.h>
void main(void)
{
    clrpsw_i();// Clears the interrupt enable bit to 0.
}
```

[Remarks]
Due to the specifications of the CLRPSW instruction used by this function, writing to the interrupt enable bit is ignored when the RX processor mode is set to user mode.
The header does not have to be included when using an intrinsic function whose name starts with __.

long macl(short *data1, short *data2, unsigned long count)
long __macl(short *data1, short *data2, unsigned long count) [V2.05.00 or later]

[Description]

Performs a multiply-and-accumulate operation between data of two bytes each and returns the result as four bytes.
The multiply-and-accumulate operation is executed with DSP functional instructions (MULLO, MACLO, and MACHI).
Data in the middle of the multiply-and-accumulate operation is retained in ACC as 48-bit data.
After all multiply-and-accumulate operations have finished, the contents of ACC are fetched by the MVFACMI instruction and used as the return value of the intrinsic function.
Usage of this intrinsic function enables fast multiply-and-accumulate operations to be expected compared to as when writing multiply-and-accumulate operations without using this intrinsic function.
This intrinsic function can be used for multiply-and-accumulate operations of 2-byte integer data. Saturation and rounding are not performed to the results of multiply-and-accumulate operations.

[Header]

<machine.h>

[Parameters]

data1 Start address of values 1 to be multiplied
data2 Start address of values 2 to be multiplied
count Count of multiply-and-accumulate operations

[Return value]

Σ(data1[n] * data2[n]) result

[Example]

```
#include <machine.h>
short data1[3] = {a1, b1, c1};
short data2[3] = {a2, b2, c2};
void mac_calc()
{
    result = macl(data1, data2, 3);
        /* Obtains the result of "a1*a2+b1*b2+c1*c2". */
}
```

[Remarks]

Refer to the programming manual to confirm the detailed contents of the various DSP functional instructions used in multiply-and-accumulate operations.
When the multiplication count is 0, the return value of the intrinsic function is 0.
When using this intrinsic function, save and restore **ACC** in an interrupt processing in which the **ACC** value is rewritten.
For the function to save and restore **ACC**, refer to the compiler option **save_acc** or the extended language specifications **#pragma interrupt**.
The header does not have to be included when using an intrinsic function whose name starts with __.

```
short macw1(short *data1, short *data2, unsigned long count)
short macw2(short *data1, short *data2, unsigned long count)
short __macw1(short *data1, short *data2, unsigned long count) [V2.05.00 or later]
short __macw2(short *data1, short *data2, unsigned long count) [V2.05.00 or later]
```

[Description]

Performs a multiply-and-accumulate operation between data of two bytes each and returns the result as two bytes.
The multiply-and-accumulate operation is executed with DSP functional instructions (MULLO, MACLO, and MACHI).
Data in the middle of the multiply-and-accumulate operation is retained in ACC as 48-bit data.
After all multiply-and-accumulate operations have finished, rounding is applied to the multiply-and-accumulate operation result of ACC.
The macw1 function performs rounding with the "RACW #1" instruction while the macw2 function performs rounding with the "RACW #2" instruction.
Rounding is performed with the following procedure.

- The contents of ACC are left-shifted by one bit with the macw1 function and by two bits with the macw2 function.
- The MSB of the lower 32 bits of ACC is rounded off (binary).
- The upper 32 bits of ACC are saturated with the upper limit as 0x00007FFF and the lower limit as 0xFFFF8000.

Finally, the contents of ACC are fetched by the MVFACHI instruction and used as the return value of these intrinsic functions.
Normally, the decimal point position of the multiplication result needs to be adjusted when fixed-point data is multiplied with each other. For example, in a case of multiplication of two Q15-format fixed-point data items, the multiplication result has to be left-shifted by one bit to make the multiplication result have the Q15 format. This left-shifting to adjust the decimal point position is achieved by the left-shift operation of the RACW instruction. Accordingly, in a case of multiply-and-accumulate operation of 2-byte fixed-point data, using these intrinsic functions facilitate multiply-and-accumulate processing. Note however that since the rounding mode of the operation result differs in macw1 and macw2, the intrinsic function to be used should be selected according to the desired accuracy for the operation result.

[Header]

<machine.h>

[Parameters]

data1	Start address of values 1 to be multiplied
data2	Start address of values 2 to be multiplied
count	Count of multiply-and-accumulate operations

[Return value]

Value obtained by rounding the multiply-and-accumulate operation result with the RACW instruction

[Example]

```
#include <machine.h>
short data1[3] = {a1, b1, c1};
short data2[3] = {a2, b2, c2};
void mac_calc()
{
    result = macw1(data1, data2, 3);
        /* Obtains the value of rounding the result of "a1*a2+b1*b2+c1*c2"*/
        /* with the "RACW #1" instruction. */
}
```

[Remarks]

Refer to the programming manual to confirm the detailed contents of the various DSP functional instructions used in multiply-and-accumulate operations.
When the multiplication count is 0, the return value of the intrinsic function is 0.
When using this intrinsic function, save and restore **ACC** in an interrupt processing in which the **ACC** value is rewritten.
For the function to save and restore **ACC**, refer to the compiler option **save_acc** or the extended language specifications **#pragma interrupt.**
The header does not have to be included when using an intrinsic function whose name starts with __.

void set_extb(void *data)
void __set_extb(void *data) [V2.05.00 or later]

[Description]
Sets a value for EXTB.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
#include <machine.h>
extern void * data;
void main(void)
{
    set_extb (data);// Sets EXTB to the value specified by data.
}
```

[Remarks]
This function is only usable when a value other than RXv1 is specified for the **isa** option or the environment variable ISA_RX. In other cases, this option will lead to an error at compilation.
Due to the specifications of the **MVTC** instruction used in this function, a write to **EXTB** is ignored when the RX processor mode is user mode.
The header does not have to be included when using an intrinsic function whose name starts with __.

void *get_extb(void)
void *__get_extb(void) [V2.05.00 or later]

[Description]
Refers to the EXTB value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
EXTB value
[Example]

```
#include <machine.h>
extern void * ret;
void main(void)
{
    ret=get_extb();// Obtains the EXTB value and stores it in ret.
}
```

[Remarks]
This function is only usable when a value other than RXv1 is specified for the **isa** option or the environment variable ISA_RX. In other cases, this option will lead to an error at compilation.
The header does not have to be included when using an intrinsic function whose name starts with __.

void __bclr(unsigned char *data, unsigned long bit) [V2.05.00 or later]

[Description]
Sets the specified one bit in the specified 1-byte area to 0 (this function is expanded into a BCLR instruction).
[Header]
-
[Parameters]
data Address of the target 1-byte area
bit Position of the bit to be manipulated
[Return value]
-
[Example]

```
unsigned char *data;
void main(void)
{
    __bclr(data, 0);  // Sets the least significant bit in the 1-byte area indicated
                      // by address data to 0.
}
```

[Remarks]
Only an integer constant from 0 to 7 can be specified as parameter bit.
This function is expanded into a BCLR instruction which directly modifies the bit specified by the parameter to 0 in memory.

void __bset(unsigned char *data, unsigned long bit) [V2.05.00 or later]

[Description]
Sets the specified one bit in the specified 1-byte area to 1 (this function is expanded into a BSET instruction).
[Header]
-
[Parameters]
data Address of the target 1-byte area
bit Position of the bit to be manipulated
[Return value]
-
[Example]

```
unsigned char *data;
void main(void)
{
    __bset(data, 0);  // Sets the least significant bit in the 1-byte area indicated
                      // by address data to 1.
}
```

[Remarks]
Only an integer constant from 0 to 7 can be specified as parameter bit.
This function is expanded into a BSET instruction which directly modifies the bit specified by the parameter to 1 in memory.

void __bnot(unsigned char *data, unsigned long bit) [V2.05.00 or later]

[Description]
Reverses the value of the specified one bit in the specified 1-byte area (this function is expanded into a BNOT instruction).
[Header]
-
[Parameters]
data Address of the target 1-byte area
bit Position of the bit to be manipulated
[Return value]
-
[Example]

```
unsigned char *data;
void main(void)
{
    __bnot(data, 0);  // Sets the least significant bit in the 1-byte area indicated
                      // by address data to 0 if the value is 1 or to 1 if the value
                      // is 0.
}
```

[Remarks]
Only an integer constant from 0 to 7 can be specified as parameter bit.
This function is expanded into a BNOT instruction which directly reverses the bit specified by the parameter in memory.

void __set_dpsw(unsigned long data) [V3.01.00 or later]

[Description]
Sets a value in DPSW.
[Header]
-
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
unsigned long data;
void main(void)
{
    __set_dpsw(data);// Sets DPSW to the value specified by data.
}
```

[Remarks]
This function is only usable when the **dpfpu** option is specified. If this call is attempted with the option not specified, an error will occur at the time of compilation.

unsigned long __get_dpsw(void) [V3.01.00 or later]

[Description]
Refers to the DPSW value.
[Header]
-
[Parameters]
-
[Return value]
DPSW value
[Example]

```
unsigned long ret;
void main(void)
{
    ret=__get_dpsw();// Obtains the DPSW value and stores it in ret.
}
```

[Remarks]
This function is only usable when the **dpfpu** option is specified. If this call is attempted with the option not specified, an error will occur at the time of compilation.

void __set_decnt(unsigned long data) [V3.01.00 or later]

[Description]
Sets a value in DECNT.
[Header]
-
[Parameters]
data Value to be set
[Return value]
-
[Example]

```
unsigned long data;
void main(void)
{
    __set_decnt(data);// Sets DECNT to the value specified by data.
}
```

[Remarks]
This function is only usable when the **dpfpu** option is specified. If this call is attempted with the option not specified, an error will occur at the time of compilation.

unsigned long __get_decnt(void) [V3.01.00 or later]

[Description]
Refers to the DECNT value.
[Header]
-
[Parameters]
-
[Return value]
DECNT value
[Example]

```
unsigned long ret;
void main(void)
{
    ret=__get_decnt();// Obtains the DECNT value and stores it in ret.
}
```

[Remarks]
This function is only usable when the **dpfpu** option is specified. If this call is attempted with the option not specified, an error will occur at the time of compilation.

void *__get_depc(void) [V3.01.00 or later]

[Description]
Refers to the DEPC value.
[Header]
-
[Parameters]
-
[Return value]
DEPC value
[Example]

```
void *ret;
void main(void)
{
    ret=__get_depc();// Obtains the DEPC value and stores it in ret.
}
```

[Remarks]
This function is only usable when the **dpfpu** option is specified. If this call is attempted with the option not specified, an error will occur at the time of compilation.

void __init_tfu(void) [V3.01.00 or later]

[Description]
Initializes the trigonometric function unit (v1).

[Header]
-

[Parameters]
-

[Return value]
-

[Example]

```
void main(void)
{
#if __TFU == 1
    __init_tfu();
#endif
}
```

[Remarks]
This function is only available when the **tfu** option is specified. If this call is attempted with the option not specified, an error will occur at the time of compilation.
When TFUv1 is used, initialize the trigonometric function unit by calling this function before use of the unit, at the startup program for example. If you do not initialize the unit, correct operation is not guaranteed.
When TFUv2 is used, there is no need to initialize the trigonometric function unit. Calling this function will lead to an error at the time of compilation.

void __sincosf(float f, float *s, float *c) [V3.01.00 or later]

[Description]
Uses the trigonometric function unit to calculate the sine and cosine of an angle at the same time (single precision).
[Header]
-
[Parameters]
f Value in radians from which to calculate the sine and cosine
s Address for storing the result of the sine operation
c Address for storing the result of the cosine operation
[Return value]
-
[Example]

```
float f, *s, *c;
void main(void)
{
#if __TFU == 1
    __init_tfu();  // The trigonometric function unit must be initialized in advance.
#endif

    __sincosf(f, s, c);
}
```

[Remarks]
This function is only available when the **tfu** option is specified. If this call is attempted with the option not specified, an error will occur at the time of compilation.
When TFUv1 is used, you must initialize the trigonometric function unit by calling the **__init_tfu()** intrinsic function before calling this function. This function is not reentrant.
When TFUv2 is used, there is no need to initialize the trigonometric function unit (calling **__init_tfu()** will lead to an error at the time of compilation). This function is reentrant by default, but is not reentrant if any of the following applies:

- When **-nosave_tfu** is specified.
- When interrupt specification **no_tfu** is specified in **#pragma interrupt** for any interrupt function that uses the trigonometric function unit.

void __atan2hypotf(float y, float x, float *a, float *h) [V3.01.00 or later]

[Description]

Uses the trigonometric function unit to calculate the arc tangent of x and y and the square root of the sum of squares of these values ($\sqrt{x^2 + y^2}$) at the same time (single precision).

[Header]

-

[Parameters]

y Coordinate y (the numerator of the tangent)
x Coordinate x (the denominator of the tangent)
a Address for storing the result of the arc tangent operation for y/x
h Address for storing the result of the square root of the sum of squares of x and y ($\sqrt{x^2 + y^2}$)

[Return value]

-

[Example]

```
float y, x, *a, *h;
void main(void)
{
#if __TFU == 1
    __init_tfu();  // The trigonometric function unit must be initialized in advance.
#endif

    __atan2hypotf(y, x, a, h);
}
```

[Remarks]

This function is only available when the **tfu** option is specified. If this call is attempted with the option not specified, an error will occur at the time of compilation.

When TFUv1 is used, you must initialize the trigonometric function unit by calling the **__init_tfu()** intrinsic function before calling this function. This function is not reentrant.

When TFUv2 is used, there is no need to initialize the trigonometric function unit (calling **__init_tfu()** will lead to an error at the time of compilation). This function is reentrant by default, but is not reentrant if any of the following applies:

- When **-nosave_tfu** is specified.
- When interrupt specification **no_tfu** is specified in **#pragma interrupt** for any interrupt function that uses the trigonometric function unit.

The result of the arc tangent operation is returned as a value in radians from -π to π.

void __sincosfx(signed long fx, signed long *s, signed long *c) [V3.05.00 or later]

[Description]
Uses the trigonometric function unit to calculate the sine and cosine of an angle at the same time (fixed-point number).
[Header]
—
[Parameters]
fx Angle used to calculate the sine and cosine
s Address for storing the result of the sine operation
c Address for storing the result of the cosine operation
[Return value]
—
[Example]

```
signed long fx, *s, *c;
void main(void) {
    __sincosfx(fx, s, c);
}
```

[Remarks]
This function is available when the **-tfu** option is specified and TFUv2 is used. If either of these conditions is not met, an error occurs at the time of compilation.
This function is reentrant by default, but is not reentrant if any of the following applies:
- When **-nosave_tfu** is specified.
- When interrupt specification **no_tfu** is specified in **#pragma interrupt** for any interrupt function that uses the trigonometric function unit.

The input format, input unit, and output format of the operation result depend on the FXSCIOC register settings. For details, refer to the hardware manual.

signed long __sinfx(signed long fx) [V3.05.00 or later]

[Description]
Uses the trigonometric function unit to calculate the sine of an angle (fixed-point number).
[[Header]
—
[Parameters]
fx Angle used to calculate the sine
[Return value]
Result of the sine operation
[Example]

```
signed long fx, s;
void main(void) {
    s = __sinfx(fx);
}
```

[Remarks]
This function is available when the **-tfu** option is specified and TFUv2 is used. If either of these conditions is not met, an error occurs at the time of compilation.
This function is reentrant by default, but is not reentrant if any of the following applies:
- When **-nosave_tfu** is specified.
- When interrupt specification **no_tfu** is specified in **#pragma interrupt** for any interrupt function that uses the trigonometric function unit.

The input format, input unit, and output format of the operation result depend on the FXSCIOC register settings. For details, refer to the hardware manual.

signed long __cosfx(signed long fx) [V3.05.00 or later]

[Description]
Uses the trigonometric function unit to calculate the cosine of an angle (fixed-point number).

[Header]
—

[Parameters]
fx Angle used to calculate the cosine

[Return value]
Result of the cosine operation

[Example]

```
signed long fx, c;
void main(void) {
    c = __cosfx(fx);
}
```

[Remarks]
This function is available when the **-tfu** option is specified and TFUv2 is used. If either of these conditions is not met, an error occurs at the time of compilation.
This function is reentrant by default, but is not reentrant if any of the following applies:

- When **-nosave_tfu** is specified.
- When interrupt specification **no_tfu** is specified in **#pragma interrupt** for any interrupt function that uses the trigonometric function unit.

The input format, input unit, and output format of the operation result depend on the FXSCIOC register settings. For details, refer to the hardware manual.

void __atan2hypotfx(signed long y, signed long x, signed long *a, signed long *h) [V3.05.00 or later]

[Description]

Uses the trigonometric function unit to calculate the arc tangent of x and y and the square root of the sum of squares of these values ($\sqrt{x^2+y^2}$) at the same time (fixed-point number).

[Header]

—

[Parameters]

y	Coordinate y (the numerator of the tangent)
x	Coordinate x (the denominator of the tangent)
a	Address for storing the result of the arc tangent operation for y/x
h	Address for storing the result of the square root of the sum of squares of x and y ($\sqrt{x^2+y^2}$)

[Return value]

—

[Example]

```
signed long y, x, *a, *h;
void main(void) {
    __atan2hypotfx(y, x, a, h);
}
```

[Remarks]

This function is available when the **-tfu** option is specified and TFUv2 is used. If either of these conditions is not met, an error occurs at the time of compilation.

This function is reentrant by default, but is not reentrant if any of the following applies:

- When **-nosave_tfu** is specified.
- When interrupt specification **no_tfu** is specified in **#pragma interrupt** for any interrupt function that uses the trigonometric function unit.

The output format and output unit of the result of the arc tangent operation depend on the FXATIOC register settings. For details, refer to the hardware manual.

The output format of the result of the square root of the sum of squares of x and y ($\sqrt{x^2+y^2}$) is fixed to Q3.29.

signed long __atan2fx(signed long y, signed long x) [V3.05.00 or later]

[Description]
Uses the trigonometric function unit to calculate the arc tangent of x and y (fixed-point number).
[Header]
—
[Parameters]
y Coordinate y (the numerator of the tangent)
x Coordinate x (the denominator of the tangent)
[Return value]
Result of the arc tangent operation for y/x
[Example]

```
signed long y, x, a;
void main(void) {
    a = __atan2fx(y, x);
}
```

[Remarks]
This function is available when the **-tfu** option is specified and TFUv2 is used. If either of these conditions is not met, an error occurs at the time of compilation.
This function is reentrant by default, but is not reentrant if any of the following applies:

- When **-nosave_tfu** is specified.
- When interrupt specification **no_tfu** is specified in **#pragma interrupt** for any interrupt function that uses the trigonometric function unit.

The output format and output unit of the operation result depend on the FXATIOC register settings. For details, refer to the hardware manual.

signed long __hypotfx(signed long x, signed long y) [V3.05.00 or later]

[Description]
Uses the trigonometric function unit to calculate the square root of the sum of squares of x and y ($\sqrt{x^2+y^2}$) (fixed-point number).
[Header]
—
[Parameters]
x Coordinate x
y Coordinate y
[Return value]
Result of the square root of the sum of squares of x and y ($\sqrt{x^2+y^2}$)
[Example]

```
signed long x, y, h;
void main(void) {
    h = __hypotfx(x, y);
}
```

[Remarks]
This function is available when the **-tfu** option is specified and TFUv2 is used. If either of these conditions is not met, an error occurs at the time of compilation.
This function is reentrant by default, but is not reentrant if any of the following applies:

- When **-nosave_tfu** is specified.
- When interrupt specification **no_tfu** is specified in **#pragma interrupt** for any interrupt function that uses the trigonometric function unit.

The output format of the operation result is fixed to Q3.29.

4.2.7 Section Address Operators

Table 4.11 lists the section address operators.

Table 4.11 Section Address Operators

No.	Section Address Operator	Description
1	__sectop("<section name>")	Refers to the start address of <section name>.
2	__secend("<section name>")	Refers to the sum of the size of <section name> and the address where <section name> starts.
3	__secsize("<section name>")	Refers to the size of <section name>.

[Return value type]
The return value type of __sectop is void *.
The return value type of __secend is void *.
The return value type of __secsize is unsigned long.

[Example]
(1) __sectop, __secend

```
#include <machine.h>
#pragma section $DSEC
static const struct {
   void *rom_s; /* Start address of the initialized data section in ROM */
   void *rom_e; /* End address of the initialized data section in ROM */
   void *ram_s; /* Start address of the initialized data section in RAM */
} DTBL[]={__sectop("D"), __secend("D"), __sectop("R")};

#pragma section $BSEC
static const struct {
   void *b_s; /* Start address of the uninitialized data section */
   void *b_e; /* End address of the uninitialized data section */
} BTBL[]={__sectop("B"), __secend("B")};

#pragma section
#pragma stacksize si=0x100
#pragma entry INIT
void main(void);
void INIT(void)
{
    _INITSCT();
    main();
    sleep();
}
```

(2) __secsize

```
/* size of section B */
unsigned int size_of_B = __secsize("B");
```

[Remarks]
In an application that enables the PIC/PID function, **__sectop** and **__secend** is processed as the addresses determined at linkage.
For details of the PIC/PID function, refer to the descriptions of the pic and pid options in Usage of PIC/PID Function.

5. ASSEMBLY LANGUAGE SPECIFICATIONS

This chapter describes the assembly language specifications supported by the RX assembler. For information on registers, instructions, and data types that are usable in the assembly source code, refer to the RX Family Software Manual.

5.1 Description of Source

This section explains description of source, expressions, and operators.

5.1.1 Description

The following shows the mnemonic line format.
[label][operation[Δoperand(s)]][comment]
Coding example:

```
LABEL1:  MOV.L      [R1], R2   ; Example of a mnemonic.
Label    Operation  Operands   Comment
```

(1) Label
Define a name for the address of the mnemonic line.

(2) Operation
Write a mnemonic or a directive.

(3) Operand(s)
Write the object(s) of the operation. The number of operands and their types depend on the operation. Some operations do not require any operands.

(4) Comment
Write notes or explanations that make the program easier to understand.

5.1.2 Names

Desired names can be defined and used in assembly-language files.
Names are classified into the following types.

Table 5.1 Types of Name

Type	Description
Label name	A name having an address as its value.
Symbol name	A name having a constant as its value (the name of a label is also included).
Section name	The name of a section that is defined through the **.SECTION** directive.
Location symbol name	The start address of the operation in a line including a location symbol ($).
Macro name	Macro definition name

Rules for Names:

- There is no limitation on the number of characters in a name.
- Names are case-sensitive; "LAB" and "Lab" are handled as different names.
- An underscore (_) and a dollar sign ($) can be used in names.
- The first character in a name must not be a digit.
- Any reserved word must not be used as a name.

Note Flag names (U, I, O, S, Z, and C), which are reserved words, can be used only for section names.

5.1.3 Coding of Labels

Be sure to append a colon (:) to the end of a label.

Example

LABEL1:

Defining a symbol name which is the same as that of an existing section is not possible. If a section and symbol with the same name are defined, the section name will be effective, but the symbol name will lead to an A2118 error.

5.1.4 Coding of Operation

- Format

mnemonic [size specifier (branch distance specifier)]

- Description

An instruction consists of the following two elements.
(1) Mnemonic: Specifies the operation of the instruction.
(2) Size specifier: Specifies the size of the data which undergoes the operation.

(1) Mnemonic

A mnemonic specifies the operation of the instruction.
Example:
`MOV`: Transfer instruction
`ADD`: Arithmetic instruction (addition instruction)

(2) Size Specifier

A size specifier specifies the size of the operand(s) in the instruction code.

- Format

.size

- Description

A size specifier specifies the operation size of the operand(s). More exactly, it specifies the size of data to be read to execute the instruction. The following can be specified as **size**.

Table 5.2 Size Specifiers

size	Description
B	Byte (8 bits)
W	Word (16 bits)
L	Longword (32 bits)

A size specifier can be written in either uppercase or lowercase.
Example: `MOV.B   #0, R3` ... Specifies the byte size.
Size specifiers can be and must be used for the instructions whose mnemonics are suffixed with "**.size**" in the Instruction Format description of the RX Family Software Manual.

(3) Branch Distance Specifier

Branch distance specifiers are used in branch and relative subroutine branch instructions.

- Format

.length

- Description

The following can be specified as **length**.

Table 5.3 Branch Distance Specifiers

length	Description	
S	3-bit PC forward relative	(+3 to +10)
B	8-bit PC relative	(–128 to +127)
W	16-bit PC relative	(–32768 to +32767)
A	24-bit PC relative	(–8388608 to +8388607)
L	Register relative	(–2147483648 to +2147183647)

A distance specifier can be written either in uppercase or lowercase.
Examples:
`BRA.W label` ... Specifies 16-bit relative.
`BRA.L R1` ... Specifies register relative.

This specifier can be omitted. When the specifier is omitted, the assembler automatically selects the distance from among **S**, **B**, **W**, and **A** to generate the smallest opcode when the following conditions are all satisfied.
(1) The operand is not a register.
(2) The operand specifies the destination for which the branch distance is determined at assembly.
Examples: Label + value determined at assembly
Label - value determined at assembly
Value determined at assembly + label
(3) The label of the operand is defined within the same section.
Note that when a register is specified as the operand, branch distance specifier **L** is selected.

For a conditional branch instruction, if the branch distance is beyond the allowed range, a code is generated by inverting the branch condition.
The following shows the branch distance specifiers that can be used in each instruction.

Table 5.4 Branch Distance Specifiers for Each Branch Instruction

Instruction		.S	.B	.W	.A	.L
BCnd	(Cnd = EQ/Z)	Allowed	Allowed	Allowed	-	-
	(Cnd = NE/NZ)	Allowed	Allowed	Allowed	-	-
	(Cnd = others)	-	Allowed	-	-	-
BRA		Allowed	Allowed	Allowed	Allowed	Allowed
BSR		-	-	Allowed	Allowed	Allowed

5.1.5 Coding of Operands

(1) Numeric Value
Five types of numeric values described below can be written in programs.
The written values are handled as 32-bit signed values (except floating-point values).

(a) Binary Number
Use digits 0 and 1, and append B or b as a suffix.

- Examples

```
1011000B
1011000b
```

(b) Octal Number
Use digits 0 to 7, and append O or o as a suffix.

- Examples

```
60702O
60702o
```

(c) Decimal Number
Use digits 0 to 9.

- Example

```
9243
```

(d) Hexadecimal Number
Use digits 0 to 9 and letters A to F and a to f, and append H or h as a suffix.
When starting with a letter, append 0 as a prefix.

- Examples

```
0A5FH
5FH
0a5fh
5fh
```

(e) Floating-Point Number
A floating-point number can be written only as the operand of the **.FLOAT** or **.DOUBLE** directive.
No floating-point number can be used in expressions.
The following range of values can be written as floating-point numbers.
FLOAT (32 bits):1.17549435 * 10^{-38} to 3.40282347 * 10^{38}
DOUBLE (64 bits):2.2250738585072014 * 10^{-308} to 1.7976931348623157 * 10^{308}

- Format

```
(mantissa)E(exponent)
(mantissa)e(exponent)
```

- Examples

```
3.4E35          ;3.4*10**35
3.4e-35         ;3.4*10**-35
-.5E20          ;-0.5*10**20
5e-20           ;5.0*10**-20
```

(2) Addressing Mode
The following three types of addressing mode can be specified in operands.

(a) General Instruction Addressing

- Register direct
The specified register is the object of operation. R0 to R15 and SP can be specified. DR0 to DR15, DRL0 to DRL15, and DRH0 to DRH15 can also be used when **-dpfpu** is specified. SP is assumed as R0 (R0 = SP).

```
Rn (Rn=R0 to R15, SP)
```

- Example:

```
ADD  R1, R2
```

- Immediate
#imm indicates an immediate integer.
#uimm indicates an immediate unsigned integer.
#simm indicates an immediate signed integer.
#imm:n, **#uimm:n**, and **#simm:n** indicate an n-bit immediate value.

```
#imm:8, #uimm:8, #simm:8, #imm:16, #simm:16, #simm:24, #imm:32
```

Note The value of **#uimm:8** in the **RTSD** instruction must be determined.

- Example:

```
MOV.L  #-100, R2; #simm:8
```

- Register indirect
  The value in the register indicates the effective address of the object of operation. The effective address range is 00000000h to FFFFFFFFh.

```
[Rn] (Rn=R0 to R15, SP)
```

- Example:

```
ADD  [R1], R2
```

- Register relative
  The effective address of the object of operation is the sum of the displacement (**dsp**) after zero-extension to 32 bits and the register value. The effective address range is 00000000h to FFFFFFFFh. **dsp:n** represents an n-bit displacement.
  Specify a **dsp** value scaled with the following rules. The assembler restores it to the value before scaling and embeds it into the instruction bit pattern.

Table 5.5 Scaling Rules of dsp Value

Instruction	Rule
Transfer instruction using a size specifier	Multiply by 1, 2, or 4 according to the size specifier (**.B**, **.W**, or **.L**)
Arithmetic/logic instruction using a size extension specifier	Multiply by 1, 1, 2, 2, or 4 according to the size extension specifier (**.B**, **.UB**, **.W**, **.UW**, or **.L**)
Bit manipulation instruction	Multiply by 1
Others	Multiply by 4

dsp:8[Rn], dsp:16[Rn] (Rn=R0 to R15, SP)

- Example:

```
ADD  400[R1], R2; dsp:8[Rn] (400/4 = 100)
```

When the size specifier is **W** or **L** but the address is not a multiple of 2 or 4:
if the value is determined at assembly: Error at assembly
if the value is not determined at assembly: Error at linkage

(b) Extended Instruction Addressing

- Short immediate
  The immediate value specified by **#imm** is the object of operation. When the immediate value is not determined at assembly, an error will be output.
  #imm:1
  This addressing mode is used only for **src** in the DSP function instruction (**RACW**). 1 or 2 can be specified as an immediate value.
  Example:

```
RACW  #1; RACW  #imm:1
```

#imm:2
The 2-bit immediate value specified by **#imm** is the object of operation. This addressing mode is only used to specify the coprocessor number in coprocessor instructions (**MVFCP**, **MVTCP**, and **OPECP**).
Example:

```
MVTCP  #3, R1, #4:16; MVTCP  #imm:2, Rn, #imm:16
```

#imm:3
The 3-bit immediate value specified by **#imm** is the object of operation. This addressing mode is used to specify the bit number in bit manipulation instructions (**BCLR**, **BMCnd**, **BNOT**, **BSET**, and **BTST**).
Example:

```
BSET   #7, R10; BSET   #imm:3, Rn
```

#imm:4
When using this addressing mode in the source statements of the **ADD**, **AND**, **CMP**, **MOV**, **MUL**, **OR**, and **SUB** instructions, the object of operation is obtained by zero-extension of the 4-bit immediate value specified by **#imm** to 32 bits.
When using this addressing mode to specify the interrupt priority level in the **MVTIPL** instruction, the object of operation is the 4-bit immediate value specified by **#imm**.
Example:

```
ADD   #15, R8; ADD   #imm:4, Rn
```

#imm:5
The 5-bit immediate value specified by **#imm** is the object of operation. This addressing mode is used to specify the bit number in bit manipulation instructions (**BCLR**, **BMCnd**, **BNOT**, **BSET**, and **BTST**), the number of bits shifted in shift instructions (**SHAR**, **SHLL**, and **SHLR**), and the number of bits rotated in rotate instructions (**ROTL** and **ROTR**).
Example:

```
BSET  #31, R10; BSET   #imm:5, Rn
```

- Short register relative
  The effective address of the object of operation is the sum of the 5-bit displacement (**dsp**) after zero-extension to 32 bits and the register value. The effective address range is 00000000h to FFFFFFFFh.
  Specify a **dsp** value respectively multiplied by 1, 2, or 4 according to the size specifier (**.B**, **.W**, or **.L**). The assembler restores it to the value before scaling and embeds it into the instruction bit pattern. When the **dsp** value is not determined at assembly, an error will be output. This addressing mode is used only in the **MOV** and **MOVU** instructions.

```
dsp:5[Rn] (Rn=R0 to R7, SP)
```

  Example:

```
MOV.L  R3,124[R1]; dsp:5[Rn] (124/4 = 31)
```

Note The other operand (**src** or **dest**) should also be R0 to R7.

- Post-increment register indirect
  1, 2, or 4 is respectively added to the register value according to the size specifier (**.B**, **.W**, or **.L**). The register value before increment is the effective address of the object of operation. The effective address range is 00000000h to FFFFFFFFh. This addressing mode is used only in the **MOV** and **MOVU** instructions.

```
[Rn+] (Rn=R0 to R15, SP)
```

  Example:

```
MOV.L [R3+],R1
```

- Pre-decrement register indirect
  1, 2, or 4 is respectively subtracted from the register value according to the size specifier (**.B**, **.W**, or **.L**). The register value after decrement is the effective address of the object of operation. The effective address range is 00000000h to FFFFFFFFh. This addressing mode is used only in the **MOV** and **MOVU** instructions.

```
[-Rn] (Rn=R0 to R15, SP)
```

  Example:

```
MOV.L [-R3],R1
```

- Indexed register indirect
  The effective address of the object of operation is the least significant 32 bits of the sum of the value in the index register (**Ri**) after multiplication by 1, 2, or 4 according to the size specifier (**.B**, **.W,** or **.L**) and the value in the base register (**Rb**). The effective address range is 00000000h to FFFFFFFFh. This addressing mode is used only in the **MOV** and **MOVU** instructions.

```
[Ri,Rb] (Ri=R0 to R15, SP)  (Rb=R0 to R15, SP)
```

Examples:

```
MOV.L [R3,R1],R2
MOV.L R3, [R1,R2]
```

(c) Specific Instruction Addressing

- Control register direct
  The specified control register is the object of operation.
  This addressing mode is used only in the **MVTC**, **POPC**, **PUSHC**, and **MVFC** instructions.

```
PSW, FPSW, USP, ISP, INTB, EXTB, BPSW, BPC, FINTV, PC
```

  This addressing mode is also used for the DPUSHM, DPOPM, MVTDC, and MVFDC instructions when **-dpfpu** is specified.

```
DPSW, DCMR, DECNT, DEPC
```

  Example:

```
STC  PSW,R2
```

- PSW direct
  The specified flag or bit is the object of operation. This addressing mode is used only in the **CLRPSW** and **SETPSW** instructions.

```
U, I, O, S, Z, C
```

  Example:

```
CLRPSW  U
```

- Program counter relative
  This addressing mode is used to specify the branch destination in the branch instruction.

```
Rn (Rn=R0 to R15, SP)
```

  The effective address is the signed sum of the program counter value and the Rn value. The range of the Rn value is –2147483648 to 2147483647. The effective address range is 00000000h to FFFFFFFFh. This addressing mode is used in the **BRA**(**.L**) and **BSR**(**.L**) instructions.

```
label(PC + pcdsp:3)
```

  This specifies the destination address of a branch instruction. The specified symbol or value indicates the effective address.
  The assembler subtracts the program counter value from the specified branch destination address and embeds it into the instruction bit pattern as a displacement (**pcdsp**).
  When the branch distance specifier is **.S**, the effective address is the least significant 32 bits of the unsigned sum of the program counter value and the displacement value.
  The range of **pcdsp** is 3 ≤ pcdsp:3 ≤ 10.
  The effective address range is 00000000h to FFFFFFFFh. This addressing mode is used only in the **BRA** and **BCnd** (only for **Cnd** == **EQ**, **NE**, **Z**, or **NZ**) instructions.

```
label(PC + pcdsp:8/pcdsp:16/pcdsp:24)
```

  This specifies the destination address of a branch instruction. The specified symbol or value indicates the effective address.
  The assembler subtracts the program counter value from the specified branch destination address and embeds it into the instruction bit pattern as a displacement (**pcdsp**).
  When the branch distance specifier is **.B**, **.W**, or **.A**, the effective address is the least significant 32 bits of the signed sum of the program counter value and the displacement value. The range of **pcdsp** is as follows.
  For **.B**: –128 ≤ pcdsp:8 ≤ +127

For **.W**: –32768 ≤ pcdsp:16 ≤ +32767
For **.A**: –8388608 ≤ pcdsp:24 ≤ +8388607
The effective address range is 00000000h to FFFFFFFFh.

(3) Bit Length Specifier
A bit length specifier specifies the size of the immediate value or displacement in the operand.

- Format

```
:width
```

- Description
This specifier should be appended immediately after the immediate value or displacement specified in the operand.
The assembler selects an addressing mode according to the specified size.
When this specifier is omitted, the assembler selects the optimum bit length for code efficiency.
When specified, the assembler does not select the optimum size but uses the specified size.
This specifier must not be used for operands of assembler directives.
One or more space characters can be inserted between an immediate value or a displacement and this specifier.
When a size specified for an instruction is not allowed for that instruction, an error will be output.
The following can be specified as **width**.
2: Indicates an effective length of one bit.
#imm:2
3: Indicates an effective length of three bits.
#imm:3
4: Indicates an effective length of four bits.
#imm:4
5: Indicates an effective length of five bits.
#imm:5, dsp:5
8: Indicates an effective length of eight bits.
#uimm:8, #simm:8, dsp:8
16: Indicates an effective length of 16 bits.
#uimm:16, #simm:16, dsp:16
24: Indicates an effective length of 24 bits.
#simm:24
32: Indicates an effective length of 32 bits.
#imm:32

(4) Size Extension Specifier
A size extension specifier specifies the size of a memory operand and the type of extension when memory is specified as the source operand of an arithmetic/logic instruction.

- Format

```
.memex
```

- Description
This specifier should be appended immediately after a memory operand and no space character should be inserted between them.
Size extension specifiers are valid only for combinations of specific instructions and memory operands; if a size extension specifier is used for an invalid combination of instruction and operand, an error will be output.
Valid combinations are indicated by "**.memex**" appended after the source operands in the Instruction Format description of the RX Family Software Manual.
When this specifier is omitted, the assembler assumes **B** for bit manipulation instructions or assumes **L** for other instructions.
The following shows available size extension specifiers and their function.

Table 5.6 Size Extension Specifiers

Size Extension Specifier	Function
B	Sign extension of 8-bit data into 32 bits
UB	Zero extension of 8-bit data into 32 bits

Size Extension Specifier	Function
W	Sign extension of 16-bit data into 32 bits
UW	Zero extension of 16-bit data into 32 bits
L	32-bit data loading

Examples:

```
ADD  [R1].B, R2
AND  125[R1].UB, R2
```

5.1.6 Expression

A combination of numeric values, symbols, and operators can be written as an expression.

- A space character or a tab can be inserted between an operator and a numeric value.
- Multiple operators can be used in combination.
- When using an expression as a symbol value, make sure that the value of the expression is determined at assembly.
- A character constant must not be used as a term of an expression.
- The expression value as a result of operation must be within the range from –2147483648 to 2147483647. The assembler does not check if the result is outside this range.

(a) Operator
The following is a list of the operators that can be written in programs.

- Unary operators

Table 5.7 Unary Operators

Operator	Function
+	Handles the value that follows the operator as a positive value.
-	Handles the value that follows the operator as a negative value.
~	Logically negates the value that follows the operator.
SIZEOF	Handles the size (bytes) of the section specified in the operand as a value.
TOPOF	Handles the start address of the section specified in the operand as a value.

Be sure to insert a space character or a tab between the operand and **SIZEOF** or **TOPOF**.
Example: `SIZEOF  program`

- Binary operators

Table 5.8 Binary Operators

Operator	Function
+	Adds the lvalue and rvalue.
-	Subtracts the rvalue from the lvalue.
*	Multiplies the lvalue and rvalue.
/	Divides the lvalue by the rvalue.
%	Obtains the remainder by dividing the lvalue by the rvalue.
>>	Shifts the lvalue to the right by the number of bits specified by the rvalue.
<<	Shifts the lvalue to the left by the number of bits specified by the rvalue.

Operator	Function
&	Logically ANDs the lvalue and rvalue in bitwise.
|	Logically (inclusively) ORs the lvalue and rvalue in bitwise.
^	Exclusively ORs the lvalue and rvalue in bitwise.

- Conditional operators
  A conditional operator can be used only in the operand of the **.IF** or **.ELIF** directive.

Table 5.9 Conditional Operators

Operator	Function
>	Evaluates if the lvalue is greater than the rvalue.
<	Evaluates if the lvalue is smaller than the rvalue.
>=	Evaluates if the lvalue is equal to or greater than the rvalue.
<=	Evaluates if the lvalue is equal to or smaller than the rvalue.
==	Evaluates if the lvalue is equal to the rvalue.
!=	Evaluates if the lvalue is not equal to the rvalue.

- Precedence designation operator

Table 5.10 Precedence Designation Operator

Operator	Function
()	An operation enclosed within ( ) takes precedence. If multiple pairs of parentheses are used in an expression, the left pair is given precedence over the right pair. Parentheses can be nested.

(b) Order of Expression Evaluation
The expression in an operand is evaluated in accordance with the following precedence and the resultant value is handled as the operand value.

- The operators are evaluated in the order of their precedence. The operator precedence is shown in the following table.
- Operators having the same precedence are evaluated from left to right.
- An operation enclosed within parentheses takes the highest precedence.

Table 5.11 Order of Expression Evaluation

Precedence	Operator Type	Operator
1	Precedence designation operator	()
2	Unary operator	+, -, ~, SIZEOF, TOPOF
3	Binary operator 1	*, /, %
4	Binary operator 2	+, -
5	Binary operator 3	>>, <<
6	Binary operator 4	&
7	Binary operator 5	|, ^
8	Conditional operator	>, <, >=, <=, ==, !=

5.1.7 Coding of Comments

A comment is written after a semicolon (;). The assembler regards all characters from the semicolon to the end of the line as a comment.

Example:

```
ADD     R1, R2          ; Adds R1 to R2.
```

5.1.8 Selection of Optimum Instruction Format

Some of the RX Family microcontroller instructions provide multiple instruction formats for an identical single processing.

The assembler selects the optimum instruction format that generates the shortest code according to the instruction and addressing mode specifications.

(1) Immediate Value

For an instruction having an immediate value as an operand, the assembler selects the optimum one of the available addressing modes according to the range of the immediate value specified as the operand. The following shows the immediate value ranges in the order of priority.

Table 5.12 Ranges of Immediate Values

#imm	Decimal Notation	Hexadecimal Notation
#imm:1	1 to 2	1H to 2H
#imm:2	0 to 3	0H to 3H
#imm:3	0 to 7	0H to 7H
#imm:4	0 to 15	0H to 0FH
#imm:5	0 to 31	0H to 1FH
#imm:8	–128 to 255	–80H to 0FFH
#uimm:8	0 to 255	0H to 0FFH
#simm:8	–128 to 127	–80H to 7FH
#imm:16	–32768 to 65535	–8000H to 0FFFFH
#simm:16	–32768 to 32767	–8000H to 7FFFH
#simm:24	–8388608 to 8388607	–800000H to 7FFFFFH
#imm:32	–2147483648 to 4294967295	–80000000H to 0FFFFFFFFH

Notes 1. Hexadecimal values can also be written in 32 bits.
Example: Decimal "–127" = hexadecimal "–7FH" can be written as "0FFFFFF81H".

Notes 2. The **#imm** range for **src** in the **INT** instruction is 0 to 255.

Notes 3. The **#imm** range for **src** in the **RTSD** instruction is four times the **#uimm:8** range.

(2) ADC and SBB Instructions

The following shows the **ADC** and **SBB** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Note The following table does not show the instruction formats and operands for which code selection is not optimized. When the processing size is not shown in the table, **L** is assumed.

Table 5.13 Instruction Formats of ADC and SBB Instructions

Instruction Format	Target of Optimum Selection			Code Size [Bytes]
	src	src2	dest	
ADC src,dest	#simm:8	-	Rd	4
	#simm:16	-	Rd	5
	#simm:24	-	Rd	6
	#imm:32	-	Rd	7
ADC/SBB src,dest	dsp:8[Rs].L	-	Rd	4
	dsp:16[Rs].L	-	Rd	5

In the **SBB** instruction, an immediate value is not allowed for **src**.

(3) ADD Instruction
The following shows the **ADD** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.14 Instruction Formats of ADD Instruction

Instruction Format	Target of Optimum Selection			Code Size [Bytes]
	src	src2	dest	
(1) ADD src,dest	#uimm:4	-	Rd	2
	#simm:8	-	Rd	3
	#simm:16	-	Rd	4
	#simm:24	-	Rd	5
	#imm:32	-	Rd	6
	dsp:8[Rs].memex	-	Rd	3 (memex = UB), 4 (memex ≠ UB)
	dsp:16[Rs].memex	-	Rd	4 (memex = UB), 5 (memex ≠ UB)
(2) ADD src,src2,dest	#simm:8	Rs	Rd	3
	#simm:16	Rs	Rd	4
	#simm:24	Rs	Rd	5
	#imm:32	Rs	Rd	6

(4) AND, OR, SUB, and MUL Instructions
The following shows the **AND**, **OR**, **SUB**, and **MUL** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.15 Instruction Formats of AND, OR, SUB, and MUL Instructions

Instruction Format	Target of Optimum Selection			Code Size [Bytes]
	src	src2	dest	
AND/OR/SUB/MUL src,dest	#uimm:4	-	Rd	2
	#simm:8	-	Rd	3
	#simm:16	-	Rd	4
	#simm:24	-	Rd	5
	#imm:32	-	Rd	6
	dsp:8[Rs].memex	-	Rd	3 (memex = UB), 4 (memex ≠ UB)
	dsp:16[Rs].memex	-	Rd	4 (memex = UB), 5 (memex ≠ UB)

In the **SUB** instruction, **#simm:8/16/24** and **#imm:32** are not allowed for **src**.

(5) BMCnd Instruction
The following shows the **BMCnd** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.16 Instruction Formats of BMCnd Instruction

Instruction Format	Processing Size	Target of Optimum Selection			Code Size [Bytes]
		src	src2	dest	
BMCnd src,dest	B	#imm:3	-	dsp:8[Rs].B	4
	B	#imm:3	-	dsp:16[Rs].B	5

(6) CMP Instruction
The following shows the **CMP** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.17 Instruction Formats of CMP Instruction

Instruction Format	Processing Size	Target of Optimum Selection			Code Size [Bytes]
		src	src2	dest	
CMP src,src2	L	#uimm:4	Rd	-	2
	L	#uimm:8	Rd	-	3
	L	#simm:8	Rd	-	3
	L	#simm:16	Rd	-	4
	L	#simm:24	Rd	-	5
	L	#imm:32	Rd	-	6
	L	dsp:8[Rs].memex	Rd	-	3 (memex = UB), 4 (memex ≠ UB)
	L	dsp:16[Rs].memex	Rd	-	4 (memex = UB), 5 (memex ≠ UB)

(7) DIV, DIVU, EMUL, EMULU, ITOF, MAX, MIN, TST, and XOR Instructions
The following shows the **DIV**, **DIVU**, **EMUL**, **EMULU**, **ITOF**, **MAX**, **MIN**, **MUL**, **TST**, and **XOR** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.18 Instruction Formats of DIV, DIVU, EMUL, EMULU, ITOF, MAX, MIN, TST, and XOR Instructions

Instruction Format	Target of Optimum Selection			Code Size [Bytes]
	src	src2	dest	
DIV/DIVU/ EMUL/EMULU/ITOF/ MAX/MIN/TST/XOR	#simm:8	-	Rd	4
	#simm:16	-	Rd	5
	#simm:24	-	Rd	6
	#imm:32	-	Rd	7
src,dest	dsp:8[Rs].memex	-	Rd	4 (memex = UB), 5 (memex ≠ UB)
	dsp:16[Rs].memex	-	Rd	5 (memex = UB), 6 (memex ≠ UB)

In the **ITOF** instruction, **#simm:8/16/24** and **#imm:32** are not allowed for **src**.

(8) FADD, FCMP, FDIV, FMUL, and FTOI Instructions
The following shows the **FADD**, **FCMP**, **FDIV**, **FMUL**, and **FTOI** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.19 Instruction Formats of FADD, FCMP, FDIV, FMUL, and FTOI Instructions

Instruction Format	Target of Optimum Selection			Code Size [Bytes]
	src	src2	dest	
FADD/FCMP/FDIV/ FMUL/FTOI src,dest	#imm:32	-	Rd	7
	dsp:8[Rs].L	-	Rd	4
	dsp:16[Rs].L	-	Rd	5

In the **FTOI** instruction, **#imm:32** is not allowed for **src**.

(9) MVTC, STNZ, and STZ Instructions
The following shows the **MVTC**, **STNZ**, and **STZ** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.20 Instruction Formats of MVTC, STNZ, and STZ Instructions

Instruction Format	Target of Optimum Selection			Code Size [Bytes]
	src	src2	dest	
MVTC/STNZ/STZ src,dest	#simm:8	-	Rd	4
	#simm:16	-	Rd	5
	#simm:24	-	Rd	6
	#imm:32	-	Rd	7

(10) MOV Instruction
The following shows the **MOV** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.21 Instruction Formats of MOV Instruction

Instruction Format	size	Processing Size	Target of Optimum Selection			Code Size [Bytes]
			src	src2	dest	
MOV(.size) src,dest	B/W/L	size	Rs (Rs=R0-R7)	-	dsp:5[Rd] (Rd=R0-R7)	2
	B/W/L	L	dsp:5[Rs] (Rs=R0-R7)	-	Rd (Rd=R0-R7)	2
	B/W/L	L	#uimm:8	-	dsp:5[Rd] (Rd=R0-R7)	3
	L	L	#uimm:4	-	Rd	2
	L	L	#uimm:8	-	Rd	3
	L	L	#simm:8	-	Rd	3
	L	L	#simm:16	-	Rd	4
	L	L	#simm:24	-	Rd	5
	L	L	#imm:32	-	Rd	6
	B	B	#imm:8	-	[Rd]	3
	W/L	W/L	#simm:8	-	[Rd]	3
	W	W	#imm:16	-	[Rd]	4
	L	L	#simm:16	-	[Rd]	4
	L	L	#simm:24	-	[Rd]	5
	L	L	#imm:32	-	[Rd]	6
	B	B	#imm:8	-	dsp:8[Rd]	4
	W/L	W/L	#simm:8	-	dsp:8[Rd]	4

Instruction Format	size	Processing Size	Target of Optimum Selection			Code Size [Bytes]
			src	src2	dest	
MOV(.size) src,dest	W	W	#imm:16	-	dsp:8[Rd]	5
	L	L	#simm:16	-	dsp:8[Rd]	5
	L	L	#simm:24	-	dsp:8[Rd]	6
	L	L	#imm:32	-	dsp:8[Rd]	7
	B	B	#imm:8	-	dsp:16[Rd]	5
	W/L	W/L	#simm:8	-	dsp:16[Rd]	5
	W	W	#imm:16	-	dsp:16[Rd]	6
	L	L	#simm:16	-	dsp:16[Rd]	6
	L	L	#simm:24	-	dsp:16[Rd]	7
	L	L	#imm:32	-	dsp:16[Rd]	8
	B/W/L	L	dsp:8[Rs]	-	Rd	3
	B/W/L	L	dsp:16[Rs]	-	Rd	4
	B/W/L	size	Rs	-	dsp:8[Rd]	3
	B/W/L	size	Rs	-	dsp:16[Rd]	4
	B/W/L	size	[Rs]	-	dsp:8[Rd]	3
	B/W/L	size	[Rs]	-	dsp:16[Rd]	4
	B/W/L	size	dsp:8[Rs]	-	[Rd]	3
	B/W/L	size	dsp:16[Rs]	-	[Rd]	4
	B/W/L	size	dsp:8[Rs]	-	dsp:8[Rd]	4
	B/W/L	size	dsp:8[Rs]	-	dsp:16[Rd]	5
	B/W/L	size	dsp:16[Rs]	-	dsp:8[Rd]	5
	B/W/L	size	dsp:16[Rs]	-	dsp:16[Rd]	6

(11) MOVU Instruction

The following shows the **MOVU** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.22 Instruction Formats of MOVU Instruction

Instruction Format	size	Processing Size	Target of Optimum Selection			Code Size [Bytes]
			src	src2	dest	
MOVU(.size) src,dest	B/W	L	dsp:5[Rs] (Rs=R0-R7)	-	Rd (Rd=R0-R7)	2
	B/W	L	dsp:8[Rs]	-	Rd	3
	B/W	L	dsp:16[Rs]	-	Rd	4

(12) PUSH Instruction

The following shows the **PUSH** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.23 Instruction Formats of PUSH Instruction

Instruction Format	Target of Optimum Selection			Code Size [Bytes]
	src	src2	dest	
PUSH src	dsp:8[Rs]	-	-	3
	dsp:16[Rs]	-	-	4

(13) ROUND Instruction
The following shows the **ROUND** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.24 Instruction Formats of ROUND Instruction

Instruction Format	Target of Optimum Selection			Code Size [Bytes]
	src	src2	dest	
ROUND src,dest	dsp:8[Rs]	-	Rd	4
	dsp:16[Rs]	-	Rd	5

(14) SCCnd Instruction
The following shows the **SCCnd** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.25 Instruction Formats of SCCnd Instruction

Instruction Format	size	Target of Optimum Selection			Code Size [Bytes]
		src	src2	dest	
SCCnd(.size) src,dest	B/W/L	-	-	dsp:8[Rd]	4
	B/W/L	-	-	dsp:16[Rd]	5

(15) XCHG Instruction
The following shows the **XCHG** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.26 Instruction Formats of XCHG Instruction

Instruction Format	Processing Size	Target of Optimum Selection			Code Size [Bytes]
		src	src2	dest	
XCHG src,dest	L	dsp:8[Rs].memex	-	Rd	4(memex = UB), 5(memex ≠ UB)
	L	dsp:16[Rs].memex	-	Rd	5(memex = UB), 6(memex ≠ UB)

(16) BCLR, BNOT, BSET, and BTST Instructions
The following shows the **BCLR**, **BNOT**, **BSET**, and **BTST** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 5.27 Instruction Formats of BCLR, BNOT, BSET, and BTST Instructions

Instruction Format	Processing Size	Target of Optimum Selection			Code Size [Bytes]
		src	src2	dest	
BCLR/BNOT/BSET/BTST src,dest	B	#imm:3	-	dsp:8[Rd].B	3
	B	#imm:3	-	dsp:16[Rd].B	4
	B	Rs	-	dsp:8[Rd].B	4
	B	Rs	-	dsp:16[Rd].B	5

5.1.9 Selection of Optimum Branch Instruction

(1) Unconditional Relative Branch (BRA) Instruction

(a) Specifiable Branch Distance Specifiers

.S 3-bit PC relative (PC + pcdsp:3, 3 ≤ pcdsp:3 ≤ 10)
.B 8-bit PC relative (PC + pcdsp:8, –128 ≤ pcdsp:8 ≤ 127)
.W 16-bit PC relative (PC + pcdsp:16, –32768 ≤ pcdsp:16 ≤ 32767)
.A 24-bit PC relative (PC + pcdsp:24, –8388608 ≤ pcdsp:24 ≤ 8388607)
.L Register relative (PC + Rs, –2147483648 ≤ Rs ≤ 2147483647)

Note The register relative distance is selected only when a register is specified as an operand; it is not used automatically through optimum selection.

(b) Optimum Selection

- The assembler selects the shortest branch distance when the operand of an unconditional relative branch instruction satisfies the conditions for optimum branch selection. For the conditions, refer to section 5.1.4 (3) Branch Distance Specifier.
- When the operand does not satisfy the conditions, the assembler selects the 24-bit PC relative distance (**.A**).

(2) Relative Subroutine Branch (BSR) Instruction

(a) Specifiable Branch Distance Specifier

.W 16-bit PC relative (PC + pcdsp:16, –32768 ≤ pcdsp:16 ≤ 32767)
.A 24-bit PC relative (PC + pcdsp:24, –8388608 ≤ pcdsp:24 ≤ 8388607)
.L Register relative (PC + Rs, –2147483648 ≤ Rs ≤ 2147483647)

Note The register relative distance is selected only when a register is specified as an operand; it is not used automatically through optimum selection.

(b) Optimum Selection

- The assembler selects the shortest branch distance when the operand of a relative subroutine branch instruction satisfies the conditions for optimum branch selection. For the conditions, refer to section 5.1.4 (3) Branch Distance Specifier.
- When the operand does not satisfy the conditions, the assembler selects the 24-bit PC relative distance (**.A**).

(3) Conditional Branch (BCnd) Instruction

(a) Specifiable Branch Distance Specifiers

BEQ.S 3-bit PC relative (PC + pcdsp:3, 3 ≤ pcdsp:3 ≤ 10)
BNE.S 3-bit PC relative (PC + pcdsp:3, 3 ≤ pcdsp:3 ≤ 10)
BCnd.B 8-bit PC relative (PC + pcdsp:8, –128 ≤ pcdsp:8 ≤ 127)
BEQ.W 16-bit PC relative (PC + pcdsp:16, –32768 ≤ pcdsp:16 ≤ 32767)
BNE.W 16-bit PC relative (PC + pcdsp:16, –32768 ≤ pcdsp:16 ≤ 32767)

(b) Optimum Selection

- When the operand of a conditional branch instruction satisfies the conditions for optimum branch selection, the assembler generates the optimum code for the conditional branch instruction by replacing it with a combination of a conditional branch instruction with an inverted logic (condition) and an unconditional relative branch instruction with an optimum branch distance.
- When the operand does not satisfy the conditions, the assembler selects the 8-bit PC relative distance (**.B**) or 16-bit PC relative distance (**.W**).

(c) Conditional Branch Instructions to Be Replaced and Corresponding Instruction Replacements

Table 5.28 Replacement Rules of Conditional Branch Instructions

Conditional Branch Instruction	Instruction Replacement	Conditional Branch Instruction	Instruction Replacement
BNC/BLTU dest	BC ..xx BRA.A dest ..xx:	BC/BGEU dest	BNC ..xx BRA.A dest ..xx:
BLEU dest	BGTU ..xx BRA.A dest ..xx:	BGTU dest	BLEU ..xx BRA.A dest ..xx:
BNZ/BNE dest	BZ ..xx BRA.A dest ..xx:	BZ/BEQ dest	BNZ ..xx BRA.A dest ..xx:
BPZ dest	BN ..xx BRA.A dest ..xx:	BO dest	BNO ..xx BRA.A dest ..xx:
BGT dest	BLE ..xx BRA.A dest ..xx:	BLE dest	BGT ..xx BRA.A dest ..xx:
BGE dest	BLT ..xx BRA.A dest ..xx:	BLT dest	BGE ..xx BRA.A dest ..xx:

Note In this table, the branch distance in unconditional relative branch instructions is a 24–bit PC relative value.
The "**..xx**" label and the unconditional relative branch instruction are processed within the assembler; only the resultant code is output to the source list file.

5.1.10 Substitute Register Names (for the PID Function)

The substitute register names listed below can be used instead of the names of general-purpose registers.

Table 5.29 Substitute Register Names

Substitute Register Name	Corresponding General-Purpose Register Name
__PID_R0	R0
__PID_R1	R1
__PID_R2	R2
__PID_R3	R3
__PID_R4	R4
__PID_R5	R5
__PID_R6	R6
__PID_R7	R7
__PID_R8	R8
__PID_R9	R9
__PID_R10	R10
__PID_R11	R11

Substitute Register Name	Corresponding General-Purpose Register Name
__PID_R12	R12
__PID_R13	R13
__PID_R14	R14
__PID_R15	R15
__PID_REG	Register selected as the PID register *1

Note *1) This indicates the name of the register selected as the PID register when the -pid or -nouse_pid_register option is specified. For details on the rules for selecting the PID register, refer to the descriptions of the -pid and -nouse_pid_register assembler options.

In assembly-language code that constitutes a master program in which the PID function is enabled, the names of all registers that may be selected as the PID register must be represented by the corresponding substitute register names (rather than the actual names of general-purpose registers such as R13).

When a substitute register name is selected as the PID register, assembling the program with nouse_pid_register enabled will not lead to an error.

[Remark]
Substitute register names are usable even when neither -nouse_pid_register nor the -pid option has been selected.

5.2 Directives

This chapter explains the directives.
Directives are instructions that direct all types of instructions necessary for the assembler.

5.2.1 Outline

Instructions are translated into object codes (machine language) as a result of assembling, but directives are not converted into object codes in principle.
Directives contain the following functions mainly:

- To facilitate description of source programs
- To initialize memory and reserve memory areas
- To provide the information required for assemblers and linkers to perform their intended processing

The following table shows the types of directives.

Type	Directives
Link directives	.SECTION, .GLB, .RVECTOR
Assembler directives	.EQU, .END, .INCLUDE
Address directives	.ORG, .OFFSET, .ENDIAN, .BLKB, .BLKW, .BLKL, .BLKD, .BYTE, .WORD, .LWORD, .FLOAT, .DOUBLE, .ALIGN
Macro directives	.MACRO, .EXITM, .LOCAL, .ENDM, .MREPEAT, .ENDR, ..MACPARA, ..MACREP, .LEN, .INSTR, .SUBSTR
Specific compiler directives	._LINE_TOP, ._LINE_END, .SWSECTION, .SWMOV, .SWITCH, .INSTALIGN

The following sections explain the details of each directive.

5.2.2 Link Directives

These directives are used for relocatable assembly that enables a program to be written in multiple separate files.

.SECTION

This directive declares or restarts a section.
[Format]

```
SECTIONΔ<section name>
.SECTIONΔ<section name>,<section attribute>
.SECTIONΔ<section name>,<section attribute>,ALIGN=[2|4|8]
.SECTIONΔ<section name>,ALIGN=[2|4|8]
<section attribute>: [CODE|ROMDATA|DATA]
```

[Description]
This directive declares or restarts a section.
(1) Declaration
This directive defines the beginning of a section with a section name and a section attribute specified.
(2) Restart
This directive specifies restart of a section that has already been declared in the source program. Specify an existing section name to restart it. The section attribute and alignment value declared before are used without change.
The alignment value in the section can be changed through the **ALIGN** specification.
The **.ALIGN** directive can be used in relative-addressing sections defined by the **.SECTION** directive including the **ALIGN** specification or in absolute-addressing sections.
When **ALIGN** is not specified, the boundary alignment value in the section is 1.
[Examples]

```
.SECTIONprogram,CODE
NOP
.SECTIONram,DATA
.BLKB10
.SECTIONtbl1,ROMDATA
.BYTE"abcd"
.SECTIONtbl2,ROMDATA,ALIGN=8
.LWORD11111111H,22222222H
.END
```

[Remarks]
Be sure to specify a section name.
To use assembler directives that allocate memory areas or store data in memory areas, be sure to define a section through this directive.
To write mnemonics, be sure to define a section through this directive.
A section attribute and **ALIGN** should be specified after a section name.
A section attribute and **ALIGN** should be specified with them separated by a comma.
A section attribute and **ALIGN** can be specified in any order.
Select **CODE**, **ROMDATA**, or **DATA** for the section attribute.
The section attribute can be omitted. In this case, the assembler assumes **CODE** as the section attribute.
When **-endian=big** is specified, only a multiple of 4 can be specified for the start address of an absolute-addressing **CODE** section.
If an absolute-addressing **CODE** section is declared when **-endian=big** is specified, a warning message will be output. In this case, the assembler appends **NOP** (0x03) at the end of the section to adjust the section size to a multiple of 4.
Defining a symbol name which is the same as that of an existing section is not possible. If a section and symbol with the same name are defined, the section name will be effective, but the symbol name will lead to an A2118 error.
The section name **$iop** is reserved and cannot be defined. If this is attempted, an A2049 error will be reported.

.GLB

This directive declares that the specified labels and symbols are global.
[Format]

```
GLBΔ<name>
.GLBΔ<name>[,<name> ?]
```

[Description]

This directive declares that the specified labels and symbols are global.

When any label or symbol specified through this directive is not defined within the current file, the assembler processes it assuming that it is defined in an external file.

When a label or symbol specified through this directive is defined within the current file, the assembler processes it so that it can be externally referenced.

[Examples]

```
.GLB     name1,name2,name3
.GLB     name4
.SECTION program
MOV.L    #name1,R1
```

[Remarks]

Be sure to insert a space character or a tab between this directive and the operand.

Specify a label name to be a global label as the operand.

Specify a symbol name to be a global symbol as the operand.

To specify multiple symbol names as operands, separate them by commas (,).

```
.RVECTOR
```

This directive registers the specified label or name as a variable vector.

[Format]

```
.RVECTORΔ<number>,<name>
```

[Description]

This directive registers the specified label or name as a variable vector.

A constant from 0 to 255 can be entered in <number> of this directive as the vector number.

A label or symbol defined within the current file can be specified as <name> of this directive.

The registered variable vectors are gathered into a single **C$VECT** section by the optimizing linkage editor.

[V3.00.00 or later] When the **-split_vect** option is specified in the optimizing linkage editor, the **C$VECT** section is split by vector table number and each section has the name of "**C$VECT**<vector table number>".

[Examples]

```
.RVECTOR   50,_rvfunc
_rvfunc:
MOV.L  #0,R1
RTE
```

[Remark]

Be sure to insert a space character or a tab between this directive and the operand.

5.2.3 Assembler Directives

These directives do not generate data corresponding to themselves but controls generation of machine code for instructions. They do not modify addresses.

```
.EQU
```

This directive defines a symbol for a 32-bit signed integer value (–2147483648 to 2147483647).

[Format]

```
<name>Δ.EQUΔ<numeric value>
```

[Description]

This directive defines a symbol for a 32-bit signed integer value (–2147483648 to 2147483647).

The symbolic debugging function can be used after symbol definition through this directive.
[Examples]

```
symbol  .EQU 1
symbol1 .EQU symbol+symbol
symbol2 .EQU 2
```

[Remarks]
The value assigned for a symbol should be determined at assembly.
Be sure to insert a space character or a tab between this directive and the operand.
A symbol can be specified as the operand of symbol definition. Note that forward-reference symbol names must not be specified.
An expression can be specified in the operand.
Symbols can be declared as global.
When this directive and the **.DEFINE** directive declare the same symbol name, the directive to make the declaration first is given priority.

.END

This directive declares the end of an assembly-language file.
[Format]

.END

[Description]
This directive declares the end of an assembly-language file.
The source file contents after the line where this directive is written are only output to the source list file; the code corresponding to them is not generated.
[Examples]

```
.END
```

[Remarks]
One **.END** directive should be written in each assembly-language file.

.INCLUDE

This directive inserts the contents of the specified include file to the line where this directive is written in the assembly-language file.
[Format]

.INCLUDEΔ<include file name>

[Description]
This directive inserts the contents of the specified include file to the line where this directive is written in the assembly-language file.
The include file contents are processed together with the contents of the assembly-language file as a single assembly-language file.
File inclusion can be nested up to 30 levels.
When an absolute path is specified as an include file name, the include file is searched for in the specified directory.
If a file is not found, an error will be output.
When the specified include file name is not an absolute path, the file is searched for in the following order.
(1) When no directory information is included in the assembly-language file name specified in the command line at assembler startup, the include file is searched for with the name specified in the **.INCLUDE** directive. When directory information is included in the assembly-language file name, the include file is searched for with the specified directory name added to the file name specified in the **.INCLUDE** directive.
(2) The directory specified through the **-include** assembler option is searched.
(3) The directory specified in the **INC_RXA** environment variable is searched.
[Examples]

```
.INCLUDE initial.src
.INCLUDE ..FILE@.inc
```

[Remarks]
Be sure to insert a space character or a tab between this directive and the operand.
Be sure to add a file extension to the include file name in the operand.
The **..FILE** directive and a string including @ can be specified as the operand.
A space character can be included in a file name, except for at the beginning of a file name.
Do not enclose a file name within double-quotes (").
The assembly-language file containing this directive cannot be specified as the include file.

5.2.4 Address Directives

These directives control address specifications in the assembler.
The assembler handles relocatable address values except for the addresses in absolute-addressing sections.

.ORG

This directive applies the absolute addressing mode to the section containing this directive.
[Format]

.ORGΔ<numeric value>

[Description]
This directive applies the absolute addressing mode to the section containing this directive.
All addresses in the section containing this directive are handled as absolute values.
This directive determines the address for storing the mnemonic code written in the line immediately after this directive.
It also determines the address of the memory area to be allocated by the area allocation directive written in the line immediately after this directive.
[Examples]

```
.SECTIONvalue,ROMDATA
.ORG0FF00H
.BYTE"abcdefghijklmnopqrstuvwxyz"
.ORG0FF80H
.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
.END
```

The following example will generate an error because **.ORG** is not written immediately after **.SECTION**.

```
.SECTIONvalue,ROMDATA
.BYTE"abcdefghijklmnopqrstuvwxyz"
.ORG0FF80H
.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
.END
```

[Remarks]
When using this directive, be sure to place it immediately after a **.SECTION** directive.
When **.ORG** is not written immediately after **.SECTION**, the section is handled as a relative-addressing section.
Be sure to insert a space character or a tab between this directive and the operand.
The operand should be a value from 0 to 0FFFFFFFFH.
An expression or a symbol can be specified as the operand. Note that the value of the expression or symbol should be determined at assembly.
This directive must not be used in a relative-addressing section.
This directive can be used multiple times in an absolute-addressing section. Note that if the value specified as the operand is smaller than the address of the line where this directive is written, an error will be output.
This directive embeds 0 or more bytes that indicate disabling (03H) for the number of addresses preceding the location specified by the offset.

.OFFSET

This directive specifies an offset from the beginning of the section.
[Format]

```
OFFSETΔ<numeric value>
```

[Description]
This directive specifies an offset from the beginning of the section.
This directive determines the offset from the beginning of the section to the area that stores the mnemonic code written in the line immediately after this directive.
It also determines the offset from the beginning of the section to the memory area to be allocated by the area allocation directive written in the line immediately after this directive.
[Examples]

```
.SECTIONvalue,ROMDATA
.BYTE"abcdefghijklmnopqrstuvwxyz"
.OFFSET80H
.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
.END
```

The following example will generate an error because the value specified in the second **.OFFSET** line is smaller than the offset to that line.

```
.SECTIONvalue,ROMDATA
.OFFSET80H
.BYTE"abcdefghijklmnopqrstuvwxyz"
.OFFSET70H
.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
.END
```

[Remarks]
Be sure to insert a space character or a tab between this directive and the operand.
The operand should be a value from 0 to 0FFFFFFFFH.
An expression or a symbol can be specified as the operand. Note that the value of the expression or symbol should be determined at assembly.
This directive must not be used in an absolute-addressing section.
This directive can be used multiple times in a relative-addressing section. Note that if the value specified as the operand is smaller than the offset to the line where this directive is written, an error will be output.
This directive embeds 0 or more bytes that indicate disabling (03H) for the number of addresses preceding the location specified by the offset.

```
.ENDIAN
```

This directive specifies the endian for the section containing this directive.
[Format]

```
.ENDIANΔBIG
.ENDIANΔLITTLE
```

[Description]
This directive specifies the endian for the section containing this directive.
When **.ENDIAN BIG** is written in a section, the byte order in the section is set to big endian.
When **.ENDIAN LITTLE** is written in a section, the byte order in the section is set to little endian.
When the directive is not written in a section, the byte order in the section depends on the **-endian** option setting.
[Examples]

```
.SECTIONvalue,ROMDATA
.ORG0FF00H
.ENDIANBIG
.BYTE"abcdefghijklmnopqrstuvwxyz"
```

The following example will generate an error because **.ENDIAN** is not written immediately after **.SECTION** or **.ORG**.

```
.SECTIONvalue,ROMDATA
.ORG0FF00H
.BYTE"abcdefghijklmnopqrstuvwxyz"
.ENDIANBIG
.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

[Remarks]
Be sure to write this directive immediately after a **.SECTION** or **.ORG** directive.
Be sure to insert a space character or a tab between this directive and the operand.
This directive must not be used in **CODE** sections.

.BLKB

This directive allocates a RAM area with the size specified in 1-byte units
[Format]

```
Δ.BLKBΔ<operand>
Δ<label name:>Δ.BLKBΔ<operand>
```

[Description]
This directive allocates a RAM area with the size specified in 1-byte units.
A label name can be defined for the address of the allocated RAM area.
[Examples]

```
symbol.EQU 1
.SECTION area,DATA
work1:.BLKB 1
work2:.BLKB symbol
.BLKB symbol+1
```

[Remarks]
Be sure to write this directive in **DATA** sections. In section definition, write ",DATA" after a section name to specify a **DATA** section.
Be sure to insert a space character or a tab between this directive and the operand.
A numeric value, a symbol, or an expression can be specified as the operand.
The operand value should be determined at assembly.
Write a label name before this directive to define the label name for the allocated area.
Be sure to append a colon (:) to the label name.
The maximum value that can be specified for the operand is 7FFFFFFFH.

.BLKW

This directive allocates 2-byte RAM areas for the specified number.
[Format]

```
Δ.BLKWΔ<operand>
Δ<label name:>Δ.BLKWΔ<operand>
```

[Description]
This directive allocates 2-byte RAM areas for the specified number.
A label name can be defined for the address of the allocated RAM area.
[Examples]

```
symbol.EQU 1
.SECTION area,DATA
work1:.BLKW 1
work2:.BLKW symbol
.BLKW symbol+1
```

[Remarks]

Be sure to write this directive in **DATA** sections. In section definition, write ",DATA" after a section name to specify a **DATA** section.
Be sure to insert a space character or a tab between this directive and the operand.
A numeric value, a symbol, or an expression can be specified as the operand.
The operand value should be determined at assembly.
Write a label name before this directive to define the label name for the allocated area.
Be sure to append a colon (:) to the label name.
The maximum value that can be specified for the operand is 3FFFFFFFH.

.BLKL

This directive allocates 4-byte RAM areas for the specified number.
[Format]

```
Δ.BLKLΔ<operand>
Δ<label name:>Δ.BLKLΔ<operand>
```

[Description]
This directive allocates 4-byte RAM areas for the specified number.
A label name can be defined for the address of the allocated RAM area.
[Examples]

```
symbol.EQU 1
.SECTION area,DATA
work1:.BLKL 1
work2:.BLKL symbol
.BLKL symbol+1
```

[Remarks]
Be sure to write this directive in **DATA** sections. In section definition, write ",DATA" after a section name to specify a **DATA** section.
Be sure to insert a space character or a tab between this directive and the operand.
A numeric value, a symbol, or an expression can be specified as the operand.
The operand value should be determined at assembly.
Write a label name before this directive to define the label name for the allocated area.
Be sure to append a colon (:) to the label name.
The maximum value that can be specified for the operand is 1FFFFFFFH.

.BLKD

This directive allocates 8-byte RAM areas for the specified number.
[Format]

```
Δ.BLKDΔ<operand>
Δ<label name:>Δ.BLKDΔ<operand>
```

[Description]
This directive allocates 8-byte RAM areas for the specified number.
A label name can be defined for the address of the allocated RAM area.
[Examples]

```
symbol.EQU 1
.SECTION area,DATA
work1:.BLKD 1
work2:.BLKD symbol
.BLKD symbol+1
```

[Remarks]
Be sure to write this directive in **DATA** sections. In section definition, write ",DATA" after a section name to specify a **DATA** section.
Be sure to insert a space character or a tab between this directive and the operand.
A numeric value, a symbol, or an expression can be specified as the operand.

The operand value should be determined at assembly.
Write a label name before this directive to define the label name for the allocated area.
Be sure to append a colon (:) to the label name.
The maximum value that can be specified for the operand is 0FFFFFFFH.

.BYTE

This directive stores 1-byte fixed data in ROM.
[Format]

```
Δ.BYTEΔ<operand>
Δ<label name:>Δ.BYTEΔ<operand>
```

[Description]
This directive stores 1-byte fixed data in ROM.
A label name can be defined for the address of the area for storing the data.
[Examples]
<When **endian=little** is specified>

```
.SECTION value,ROMDATA
.BYTE 1
.BYTE "data"
.BYTE symbol
.BYTE symbol+1
.BYTE 1,2,3,4,5
.END
```

<When **endian=big** is specified>

```
.SECTION  program,CODE,ALIGN=4
MOV.L  R1,R2
.ALIGN 4
.BYTE 080H,00H,00H,00H
.END
```

[Remarks]
Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.
Be sure to insert a space character or a tab between this directive and the operand.
A numeric value, a symbol, or an expression can be specified as the operand.
To specify a character or a string for the operand, enclose it within single-quotes (') or double-quotes ("). In this case, the ASCII code for the specified characters is stored.
Write a label name before this directive to define the label name for the area storing the data.
Be sure to append a colon (:) to the label name.
When the **endian=big** option is specified, this directive can be used only in the sections that satisfy the following conditions. An error will be output if this directive is used in a section that does not satisfy the conditions.

(1) **ROMDATA** section

```
.SECTION data,ROMDATA
```

(2) Relative-addressing **CODE** section for which the address alignment value is set to 4 or 8 in section definition

```
.SECTION program,CODE,ALIGN=4
```

(3) Absolute-addressing **CODE** section

```
.SECTION program,CODE
.ORG 0fff00000H
```

To use a **.BYTE** directive in a **CODE** section while the **endian=big** option is specified, be sure to write an address correction directive (**.ALIGN 4**) in the line immediately before the **.BYTE** directive so that the data is aligned to a 4-byte boundary. If this address correction directive is not written, the assembler outputs a warning message and automatically aligns the data to a 4-byte boundary.

When the **endian=big** option is specified, the data area size in a **CODE** section must be specified to become a multiple of 4. If the data area size in a **CODE** section is not a multiple of 4, the assembler outputs a warning message and writes **NOP** (0x03) to make the data area size become a multiple of 4.

.WORD

This directive stores 2-byte fixed data in ROM.

[Format]

```
Δ.WORDΔ<operand>
Δ<label name:>Δ.WORDΔ<operand>
```

[Description]

This directive stores 2-byte fixed data in ROM.

A label name can be defined for the address of the area for storing the data.

[Examples]

```
.SECTION value,ROMDATA
.WORD1
.WORDsymbol
.WORDsymbol+1
.WORD1,2,3,4,5
.END
```

[Remarks]

Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.

Be sure to insert a space character or a tab between this directive and the operand.

A numeric value, a symbol, or an expression can be specified as the operand.

Neither a character nor a string can be specified for an operand.

Write a label name before this directive to define the label name for the area storing the data.

Be sure to append a colon (:) to the label name.

.LWORD

This directive stores 4-byte fixed data in ROM.

[Format]

```
Δ.LWORDΔ<operand>
Δ<label name:>Δ.LWORDΔ<operand>
```

[Description]

This directive stores 4-byte fixed data in ROM.

A label name can be defined for the address of the area for storing the data.

```
.SECTION value,ROMDATA
.LWORD1
.LWORDsymbol
.LWORDsymbol+1
.LWORD1,2,3,4,5
.END
```

[Remarks]

Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.

Be sure to insert a space character or a tab between this directive and the operand.

A numeric value, a symbol, or an expression can be specified as the operand.

Neither a character nor a string can be specified for an operand.

Write a label name before this directive to define the label name for the area storing the data.

Be sure to append a colon (:) to the label name.

.FLOAT

This directive stores 4-byte fixed data in ROM.
[Format]

Δ.FLOATΔ<numeric value>
Δ<label name:>Δ.FLOATΔ<numeric value>

[Description]
This directive stores 4-byte fixed data in ROM.
A label name can be defined for the address of the area for storing the data.
[Examples]

```
          .FLOAT  5E2
constant: .FLOAT  5e2
```

[Remarks]
Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.
Specify a floating-point number as the operand.
Be sure to insert a space character or a tab between this directive and the operand.
Write a label name before this directive to define the label name for the area storing the data.
Be sure to append a colon (:) to the label name.

.DOUBLE

This directive stores 8-byte fixed data in ROM.
[Format]

Δ.DOUBLEΔ<numeric value>
Δ<label name:>Δ.DOUBLEΔ<numeric value>

[Description]
This directive stores 8-byte fixed data in ROM.
A label name can be defined for the address of the area for storing the data.
[Examples]

```
          .DOUBLE  5E2
constant: .DOUBLE  5e2
```

[Remarks]
Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.
Specify a floating-point number as the operand.
Be sure to insert a space character or a tab between this directive and the operand.
Write a label name before this directive to define the label name for the area storing the data.
Be sure to append a colon (:) to the label name.

.ALIGN

This directive corrects the address for storing the code written in the line immediately after this directive to a multiple of two, four, or eight bytes.
[Format]

Δ.ALIGNΔ<alignment value>
<alignment value>: [2|4|8]

[Description]
This directive corrects the address for storing the code written in the line immediately after this directive to a multiple of two, four, or eight bytes.

In a **CODE** or **ROMDATA** section, **NOP** code (03H) is written to the empty space generated as a result of address correction.

In a **DATA** section, only address correction is performed.

[Examples]

```
.SECTION program,CODE,ALIGN=4
MOV.L  R1, R2
.ALIGN  4; Corrects the address to a multiple of 4
RTS
.END
```

[Remarks]

This directive can be used in the sections that satisfy the following conditions.

(1) Relative-addressing section for which address correction is specified in section definition

```
.SECTION program,CODE,ALIGN=4
```

(2) Absolute-addressing section

```
.SECTION program,CODE
.ORG 0fff00000H
```

A warning message will be output if this directive is used for a relative-addressing section in which **ALIGN** is not specified in the **.SECTION** directive line.

A warning message will be output if the specified value is larger than the boundary alignment value specified for the section.

5.2.5 Macro Directives

These directives do not generate data corresponding to themselves but controls generation of machine code for instructions. They do not modify addresses.

These directives define macro functions and repeat macro functions.

Table 5.30 Macro Directives

Directive	Function
.MACRO	Defines a macro name and the beginning of a macro body.
.EXITM	Terminates macro body expansion.
.LOCAL	Declares a local label in a macro.
.ENDM	Specifies the end of a macro body.
.MREPEAT	Specifies the beginning of a repeat macro body.
.ENDR	Specifies the end of a repeat macro body.
..MACPARA	Indicates the number of arguments in a macro call.
..MACREP	Indicates the count of repeat macro body expansions.
.LEN	Indicates the number of characters in a specified string.
.INSTR	Indicates the start position of a specified string in another specified string.
.SUBSTR	Extracts a specified number of characters from a specified position in a specified string.

.MACRO

This directive defines a macro name.

[Format]

```
[macro definition]
Δ<macro name>Δ.MACRO[<parameter>[,...]]
Δbody
Δ.ENDM
[macro call]
Δ<macro name>Δ[<argument>[,...]]
```

[Description]
This directive defines a macro name.
It also specifies the beginning of a macro definition.
[Examples: Example 1]
[Macro definition example]

```
name.MACRO  string
.BYTE  'string'
.ENDM
```

[Macro call example 1]

```
name"name,address"

.BYTE'name,address'
```

[Macro call example 2]

```
name(name,address)

.BYTE'(name,address)'
```

[Example 2]

```
mac    .MACROp1,p2,p3
.IF   ..MACPARA == 3
.IF   'p1' == 'byte'
MOV.B  #p2,[p3]
.ELSE
MOV.W  #p2,[p3]
.ENDIF
.ELIF..MACPARA == 2
.IF   'p1' == 'byte'
MOV.B  #p2,[R3]
.ELSE
MOV.W  #p2,[R3]
.ENDIF
.ELSE
MOV.W  R3,R1
.ENDIF
.ENDM

mac word,10,R3; Macro call

.IF   3 == 3; Macro-expanded code
.ELSE
MOV.W  #10,[R3]
.ENDIF
```

[Remarks]
Be sure to specify a macro name.
For the macro name and parameter name format, refer to the Rules for Names in section 4.1.2, Names.
Use a unique name for defining each parameter, including the nested macro definitions.
To define multiple parameters, separate them by commas (,).
Make sure that all parameters specified as operands of a **.MACRO** directive are used in the macro body.
Be sure to insert a space character or a tab between a macro name and an argument.

Write a macro call so that the arguments correspond to the parameters on a one-to-one basis.
To use a special character in an argument, enclose it within double-quotes.
A label, a global label, and a symbol can be used in an argument.
An expression can be used in an argument.
Parameters are replaced with arguments from left to right in the order they appear.
If no argument is specified in a macro call while the corresponding parameter is defined, the assembler does not generate code for this parameter.
If there are more parameters than the arguments, the assembler does not generate code for the parameters that do not have the corresponding arguments.
When a parameter in the body is enclosed within single-quotes ('), the assembler encloses the corresponding argument within single-quotes when outputting it.
When an argument contains a comma (,) and the argument is enclosed within parentheses (( )), the assembler converts the argument including the parentheses.
If there are more arguments than the parameters, the assembler does not process the arguments that do not have the corresponding parameters.
The string enclosed within double-quotes is processed as a string itself. Do not enclose parameters within double-quotes.
Up to 80 parameters can be specified within the maximum allowable number of characters for one line.
If the number of arguments differs from that of the parameters, the assembler outputs a warning message.

.EXITM

This directive terminates expansion of a macro body and passes control to the nearest **.ENDM**.
[Format]

```
<macro name>Δ.MACRO
Δbody
Δ.EXITM
Δbody
Δ.ENDM
```

[Description]
This directive terminates expansion of a macro body and passes control to the nearest **.ENDM**.
[Examples]

```
data1  .MACROvalue
.IF   value == 0
       .EXITM
.ELSE
.BLKBvalue
.ENDIF
.ENDM

data1  0  ; Macro call

.IF      0 == 0; Macro-expanded code
.EXITM
.ENDIF
```

[Remarks]
Write this directive in the body of a macro definition.

.LOCAL

This directive declares that the label specified as an operand is a macro local label.
[Format]

```
.LOCALΔ<label name>[,...]
```

[Description]
This directive declares that the label specified as an operand is a macro local label.

Macro local labels can be specified multiple times with the same name as long as they are specified in different macro definitions or outside macro definitions.

[Examples]

```
name.MACRO
.LOCALm1; 'm1' is macro local label
m1:
 nop
 bram1
 .ENDM
```

[Remarks]

Write this directive in a macro body.

Be sure to insert a space character or a tab between this directive and the operand.

Make sure that a macro local label is declared through this directive before the label name is defined.

For the macro local name format, refer to the Rules for Names in section 10.1.2, Names.

Multiple labels can be specified as operands of this directive by separating them by commas. Up to 100 labels can be specified in this manner.

When macro definitions are nested, a macro local label in a macro that is defined within another macro definition (outer macro) cannot use the same name as that used in the outer macro.

Up to 65,535 macro local labels can be written in one assembly source file including those used in the include files.

.ENDM

This directive specifies the end of a macro definition.

[Format]

```
<macro name>Δ.MACRO
Δbody
Δ.ENDM
```

[Description]

This directive specifies the end of a macro definition.

[Examples]

```
lda  .MACRO
MOV.L  #value,R3
.ENDM
lda  0  ; Expanded to MOV.L  #0,R3.
```

.MREPEAT

This directive specifies the beginning of a repeat macro.

[Format]

```
[<label>:]Δ.MREPEATΔ<numeric value>
Δbody
Δ.ENDR
```

[Description]

This directive specifies the beginning of a repeat macro.

The assembler repeatedly expands the body the specified number of times.

The repetition count can be specified within the range of 1 to 65,535.

Repeat macros can be nested up to 65,535 levels.

The macro body is expanded at the line where this directive is written.

[Examples]

```
rep  .MACRO  num
.MREPEAT num
.IF   num > 49
.EXITM
.ENDIF
nop
.ENDR
 .ENDM

rep  3  ; Macro call

nop     ; Macro-expanded code
nop
nop
```

[Remarks]
Be sure to specify an operand.
Be sure to insert a space character or a tab between this directive and the operand.
A label can be specified at the beginning of this directive line.
A symbol can be specified as the operand.
Forward reference symbols must not be used.
An expression can be used in the operand.
Macro definitions and macro calls can be used in the body.
The **.EXITM** directive can be used in the body.

.ENDR

This directive specifies the end of a repeat macro.
[Format]

```
[<label>:]Δ.MREPEATΔ<numeric value>
Δbody
Δ.ENDR
```

[Description]
This directive specifies the end of a repeat macro.
[Remarks]
Make sure this directive corresponds to an **.MREPEAT** directive.

..MACPARA

This directive indicates the number of arguments in a macro call.
[Format]

```
..MACPARA
```

[Description]
This directive indicates the number of arguments in a macro call.
This directive can be used in the body in a macro definition through **.MACRO**.
[Examples]
This example executes conditional assembly according to the number of macro arguments.

```
        .GLBmem
name.MACRO  f1,f2
        .IF..MACPARA == 2
        ADD  f1,f2
        .ELSE
        ADD  R3,f1
        .ENDIF
        .ENDM

        name  mem    ; Macro call

        .ELSE        ; Macro-expanded code
        ADD  R3,mem
        .ENDIF
```

[Remarks]
This directive can be used as a term of an expression.
If this directive is written outside a macro body defined through **.MACRO**, its value becomes 0.

..MACREP

This directive indicates the count of repeat macro expansions.
[Format]

```
..MACREP
```

[Description]
This directive indicates the count of repeat macro expansions.
This directive can be used in the body in a macro definition through **.MREPEAT**.
This directive can be specified in an operand of conditional assembly.
[Examples]

```
mac .MACRO  value,reg
        .MREPEAT  value
        MOV.B#0,..MACREP[reg]
        .ENDR
        .ENDM

        mac3,R3; Macro call

        .MREPEAT3; Macro-expanded code
        MOV.B#0,1[R3]
        MOV.B#0,2[R3]
        MOV.B#0,3[R3]
        .ENDR
        .ENDM
```

[Remarks]
This directive can be used as a term of an expression.
If this directive is written outside a macro body defined through **.MACRO**, its value becomes 0.

.LEN

This directive indicates the length of the string specified as the operand.
[Format]

```
.LENΔ{"<string>"}
.LENΔ{'<string>'}
```

[Description]
This directive indicates the length of the string specified as the operand.

[Examples]

```
bufset.MACRO  f1
buffer:.BLKB  .LEN{'f1'}
.ENDM

bufset  Sample     ; Macro call

buffer:.BLKB   6   ; Macro-expanded code
```

[Remarks]
Be sure to enclose the operand within {}.
A space character or a tab can be inserted between this directive and the operand.
Characters including spaces and tabs can be specified in a string.
Be sure to enclose a string within single-quotes or double-quotes.
This directive can be used as a term of an expression.
To count the length of the macro argument, enclose the parameter name within single-quotes. When the name is enclosed within double-quotes, the length of the string specified as the parameter is counted.

.INSTR

This directive indicates the start position of a search string within a specified string.
[Format]

```
.INSTRΔ{"<string>","<search string>",<search start position> }
.INSTRΔ{'<string>','<search string>',<search start position> }
```

[Description]
This directive indicates the start position of a search string within a specified string.
The position from which search is started can be specified.
[Examples]
This example detects the position (7) of string "**se**", counted from the beginning (**top**) of a specified string (**japanese**):

```
top  .EQU   1
point_set.MACRO  source,dest,top
point.EQU    .INSTR{'source','dest',top}
.ENDM
point_set  japanese,se,1   ; Macro call

point       .EQU   7      ; Macro-expanded code
```

[Remarks]
Be sure to enclose the operand within {}.
Be sure to specify all of a string, a search string, and a search start position.
Separate the string, search string, and search start position by commas.
Neither space character nor tab can be inserted before or after a comma.
A symbol can be specified as a search start position.
When 1 is specified as the search start position, it indicates the beginning of a string.
This directive can be used as a term of an expression.
This directive is replaced with 0 when the search string is longer than the string, the search string is not found in the string, or the search start position value is larger than the length of the string.
To expand a macro by using a macro argument as the condition for detection, enclose the parameter name within single-quotes. When the name is enclosed within double-quotes, the macro is expanded by using the enclosed string as the condition for detection.

.SUBSTR

This directive extracts a specified number of characters from a specified position in a specified string.
[Format]

```
.SUBSTRΔ{"<string>",<extraction start position>,<extraction character length> }
.SUBSTRΔ{ '<string>',<extraction start position>,<extraction character length> }
```

[Description]
This directive extracts a specified number of characters from a specified position in a specified string.
[Examples]
The following example passes the length of the string given as an argument of a macro to the operand of **.MREPEAT**.
The **..MACREP** value is incremented as 1 -> 2 -> 3 -> 4 every time the **.BYTE** line is expanded. Consequently, the characters in the string given as an argument of the macro is passed to the operand of **.BYTE** one by one starting from the beginning of the string.

```
name.MACRO  data
.MREPEAT.LEN{'data'}
.BYTE.SUBSTR{'data',..MACREP,1}
.ENDR
.ENDM

name  ABCD     ; Macro call

.BYTE  "A"     ; Macro-expanded code
.BYTE  "B"
.BYTE  "C"
.BYTE  "D"
```

[Remarks]
Be sure to enclose the operand within {}.
Be sure to specify all of a string, an extraction start position, and an extraction character length.
Separate the string, extraction start position, and extraction character length by commas.
Symbols can be specified as an extraction start position and an extraction character length. When 1 is specified as the extraction start position, it indicates the beginning of a string.
Characters including spaces and tabs can be specified in a string.
Be sure to enclose a string within single-quotes or double-quotes.
This directive is replaced with 0 when the extraction start position value is larger than the string, the extraction character length is larger than the length of the string, or the extraction character length is set to 0.
To expand a macro by using the macro argument as the condition for extraction, enclose the parameter name within single-quotes. When the name is enclosed within double-quotes, the macro is expanded by using the enclosed string as the condition for extraction.

5.2.6 Specific Compiler Directives

The following directives are output in some cases so that the assembler can appropriately process C language functions when the compiler generates assembly-language files.
When using the assembly-language files generated by the compiler, these directives should be used without changing the settings. These directives should not be used when creating user-created assembly-language files.

Table 5.31 Specific Compiler Directives

Directive	Function
._LINE_TOP	These directives are output when the functions specified by **#pragma inline_asm** have been expanded.
._LINE_END	
.SWSECTION	These directives are output when the branch table is used in the **switch** statement.
.SWMOV	
.SWITCH	
.INSTALIGN	This directive is output when the **instalign4** option, the **instalign8** option, **#pragma instalign4,** or **#pragma instalign8** is used.

5.3 Control Instructions

This chapter describes control instructions.
Control Instructions provide detailed instructions for assembler operation.

5.3.1 Outline

Control instructions provide detailed instructions for assembler operation and so are written in the source.
Control instructions do not become the target of object code generation.
The following table shows the types of control instructions.

Type	Control Instructions
Assembler list directive	.LIST
Conditional assembly directives	.IF, .ELIF, .ELSE, .ENDIF
Extended function directives	.ASSERT, ?, @, ..FILE, .STACK, .LINE, .DEFINE

The following sections explain the details of each control instruction.

5.3.2 Assembler List Directive

This directive controls the output information and format of the assembler list file. It does not affect code generation.

```
.LIST
```

This directive can stop (**OFF**) outputting lines to the assembler list file.
[Format]

```
.LIST∆[ON|OFF]
```

[Description]
This directive can stop (**OFF**) outputting lines to the assembler list file.
Even in the range where line output is stopped, error lines are output to the assembler list file.
This directive can start (**ON**) outputting lines to the assembler list file.
When this directive is not specified, all lines are output to the assembler list file.
[Examples]

```
.LIST  ON
.LIST  OFF
```

[Remarks]
Be sure to insert a space character or a tab between this directive and the operand.
Specify **OFF** as the operand to stop outputting lines.
Specify **ON** as the operand to start outputting lines.

5.3.3 Conditional Assembly Directives

These directives specify whether to assemble a specified range of lines.

Table 5.32 Conditional Assembly Directives

Directive	Function
.IF	Specifies the beginning of a conditional assembly block and evaluates the condition.
.ELIF	Evaluates the second or later conditions when multiple conditional blocks are written.
.ELSE	Specifies the beginning of a block to be assembled when all conditions are false.
.ENDIF	Specifies the end of a conditional assembly block.

.IF, .ELIF, .ELSE, .ENDIF

[Format]

```
.IFΔconditional expression
body
.ELIFΔconditional expression
body
.ELSE
body
.ENDIF
```

[Description]

The assembler controls assembly of the blocks according to the conditions specified through **.IF** and **.ELIF**.

The assembler evaluates the condition specified in the operand of **.IF** or **.ELIF**, and assembles the body in the subsequent lines when the condition is true. In this case, the lines before the **.ELIF**, **.ELSE**, or **.ENDIF** directive are assembled.

Any directives that can be used in an assembly-language file can be written in a conditional assembly block.

Conditional assembly is done according to the result of conditional expression evaluation.

[Examples] <Example of conditional expressions>

```
sym < 1
sym+2 < data1
sym+2 < data1+2
'smp1' == name
```

<Example of conditional assembly specification>

```
.IF    TYPE==0
.byte "Proto Type Mode"
.ELIF TYPE>0
.byte "Mass Production Mode"
.ELSE
.byte "Debug Mode"
.ENDIF
```

[Remarks]

Be sure to write a conditional expression in an **.IF** or **.ELIF** directive.

Be sure to insert a space character or a tab between the **.IF** or **.ELIF** directive and the operand.

Only one conditional expression can be specified for the operand of the **.IF** or **.ELIF** directive.

Be sure to use a conditional operator in a conditional expression.

The following operators can be used.

Table 5.33 Conditional Operators of .IF and .ELIF Directives

Conditional Operator	Description
>	The condition is true when the lvalue is greater than the rvalue
<	The condition is true when the lvalue is smaller than the rvalue
>=	The condition is true when the lvalue is equal to or greater than the rvalue
<=	The condition is true when the lvalue is equal to or smaller than the rvalue
==	The condition is true when the lvalue is equal to the rvalue
!=	The condition is true when the lvalue is not equal to the rvalue

A conditional expression is evaluated in signed 32 bits.

Symbols can be used in the left and right sides of a conditional operator.

Expressions can be used in the left and right sides of a conditional operator. For the expression format, refer to the rules described in (2) Expression in section 4.1.5, Coding of Operands.

Strings can be used in the left and right sides of a conditional operator. Be sure to enclose a string within single-quotes (') or double-quotes ("). Strings are compared in character code values.

Examples:
`"ABC"<"CBA"` -> 414243 < 434241; this condition is true.
`"C" < "A"` -> 43 < 41; this condition is false.
Space characters and tabs can be written before and after conditional operators.
Conditional expressions can be specified in the operands of the **.IF** and **.ELIF** directives.
The assembler does not check if the evaluation result is outside the allowed range.
Forward reference symbols (reference to a symbol that is defined after this directive line) must not be specified.
If a forward reference symbol or an undefined symbol is specified, the assembler assumes the symbol value as 0 when evaluating the expression.

5.3.4 Extended Function Directives

These directives do not affect code generation.

Table 5.34 Extended Function Directives

Directive	Function
.ASSERT	Outputs a string specified in an operand to the standard error output or a file.
?	Defines and references a temporary label.
@	Concatenates strings specified before and after @ so that they are handled as one string.
..FILE	Indicates the name of the assembly-language file being processed by the assembler.
.STACK	Defines a stack value for a specified symbol.
.LINE	Changes line number.
.DEFINE	Defines a replacement symbol.

.ASSERT

This directive outputs a string specified in the operand to the standard error output at assembly.
[Format]

```
.ASSERTΔ"<string>"
.ASSERTΔ"<string>">Δ<file name>
.ASSERTΔ"<string>">>Δ<file name>
```

[Description]
This directive outputs a string specified in the operand to the standard error output at assembly.
When a file name is specified, the assembler outputs the string written in the operand to the file.
When an absolute path is specified as a file name, the assembler creates a file in the specified directory.
When no absolute path is specified as a file name;
(1) if no directory information is included in the file name specified by the **output** option, the assembler creates the file specified by this directive in the current directory.
(2) if directory information is included in the file name specified by the **output** option, the assembler creates the file specified by this directive and adds the directory information for the file specified by the **output** option.
(3) if the **output** option is not specified, the assembler creates the file in the same directory containing the file specified in the command line at assembler startup.
When the **..FILE** directive is specified as a file name, the assembler creates a file in the same directory as the file specified in the command line at assembler startup.
[Examples]

To output a message to the sample.dat file:
.ASSERT "string" > sample.dat
To add a message to the sample.dat file:
.ASSERT "string" >> sample.dat
To output a message to a file with the same name as the current processing file but without a file extension:
.ASSERT "string" > ..FILE

[Remarks]

Be sure to insert a space character or a tab between the directive and the operand.
Be sure to enclose the string in the operand within double-quotes.
To output a string to a file, specify the file name after > or >>.
The symbol > directs the assembler to create a new file and output a message to the file. If a file with the same name exists, the file is overwritten.
The symbol >> directs the assembler to add the message to the contents of the specified file. If the specified file is not found, the assembler creates a new file.
Space characters or tabs can be specified before and after > and >>.
The **..FILE** directive can be specified as a file name.

?

This directive defines a temporary label.
[Format]

```
?:
Δ<mnemonic >Δ?+
Δ<mnemonic >Δ?-
```

[Description]
This directive defines a temporary label.
It also references the temporary label defined immediately before or after an instruction.
Definitions and references are allowed within the same file.
Up to 65,535 temporary labels can be defined in a file. In this case, if **.INCLUDE** is used in the file, the maximum number (65,535) of temporary files includes the labels in the include file.
The temporary labels converted by the assembler are output to the source list file.
[Examples]

```
?:
 BRA  ?+
 BRA  ?-
?:
 BRA  ?-
```

References temporary labels indicated by arrows.

[Remarks]
Write "?:" in the line that is to be defined as a temporary label.
To reference the temporary label defined immediately before an instruction, write "?-" as an operand of the instruction.
To reference the temporary label defined immediately after an instruction, write "?+" as an operand of the instruction.
Only the label defined immediately before or after an instruction can be referenced from the instruction.

@

This directive concatenates macro arguments, macro variables, reserved symbols, an expanded file name of directive **..FILE**, and specified strings.
[Format]

```
<string>@<string>[@<string> ...]
```

[Description]
This directive concatenates macro arguments, macro variables, reserved symbols, an expanded file name of directive **..FILE**, and specified strings.
[Examples]
Example of file name concatenation:
When the name of the currently processed file is **sample1.src**, a message is output to the **sample.dat** file in the following example.

```
.ASEERT   "sample" > ..FILE@.dat
```

Example of string concatenation:

```
mov_nibble .MACRO p1,src,dest
MOV.@p1 src,dest
.ENDM

mov_nibble  W,R1,R2; Macro call

MOV.W R1,R2 ;Macro-expanded code
```

[Remarks]
Space characters and tabs inserted before and after this directive are concatenated as a string.
Strings can be written before and after this directive.
To use @ as character data (40H), enclose it within double-quotes ("). When a string including @ is enclosed within single-quotes ('), the strings before and after @ are concatenated.
This directive can be used multiple times in one line.
To use the concatenated string as a name, do not insert space characters or tabs before or after this directive.

..FILE

This directive is expanded to the name of the file that the assembler is currently processing (assembly-language file name or include file name).
[Format]

..FILE

[Description]
This directive is expanded to the name of the file that the assembler is currently processing (assembly-language file name or include file name).
[Examples]
When the assembly-language file name is **sample.src**, a message is output to the **sample** file in the following example.

```
.ASSERT "sample" > ..FILE
```

When the assembly-language file name is **sample.src**, the **sample.inc** file is included in the following example.

```
.INCLUDE ..FILE@.inc
```

When the above line is written in the **incl.inc** file included in the **sample.src** file, a string is output to the **incl.mes** file in most cases.

```
.ASSERT "sample" > ..FILE@.mes
```

[Remarks]
This directive can be used in the operand of the **.ASSERT** and **.INCLUDE** directives.
Only the file name body with neither file extension nor path is used for replacement.

.STACK

This directive defines the stack size to be used for a specified symbol referenced through the Call Walker.
[Format]

.STACKΔ<name>=<numeric value>

[Description]
This directive defines the stack size to be used for a specified symbol referenced through the Call Walker.
[Examples]

```
.STACK  SYMBOL=100H
```

[Remarks]

The stack value for a symbol can be defined only once; any later definitions for the same symbol are ignored. A multiple of 4 in the range from 0H to 0FFFFFFFCH can be specified for a stack value, and a definition with any other value is ignored.

<numeric value> must be a constant specified without using a forward reference symbol, an externally referenced symbol, or a relative address symbol.

.LINE

This directive changes the line number and file name referred to in assembler error messages or at debugging.

[Format]

```
.LINEΔ<file name>,<line number>
.LINEΔ<line number>
```

[Description]

This directive changes the line number and file name referred to in assembler error messages or at debugging.

The line number and the file name specified with **.LINE** are valid until the next **.LINE** in a program.

The compiler generates **.LINE** corresponding to the line in the C source file when the assembly source program is output with the debugging option specified.

When the file name is omitted, the file name is not changed, but only the line number is changed.

[Examples]

```
.LINE  "C:\asm\test.c",5
```

.DEFINE

This directive defines a symbol for a string.

[Format]

```
<symbol name>Δ.DEFINEΔ<string>
<symbol name>Δ.DEFINEΔ'<string>'
<symbol name>Δ.DEFINEΔ"<string>"
```

[Description]

This directive defines a symbol for a string. Defined symbols can be redefined.

[Examples]

```
X_HI.DEFINE   R1
MOV.L  #0, X_HI
```

[Remarks]

To define a symbol for a string including a space character or a tab, be sure to enclose it within single-quotes (') or double-quotes (").

The symbols defined through this directive cannot be declared as external references.

When this directive and the **.EQU** directive declare the same symbol name, the directive to make the declaration first is given priority.

5.4 Macro Names

The following predefined macros are defined according to the option specification and version.

Table 5.35 Predefined Macros of Assembler

	Macro Name	Value	Option
1	__RX600 __RX200	.DEFINE 1 .DEFINE 1	cpu=rx600 cpu=rx200
2	__BIG __LITTLE	.DEFINE 1 .DEFINE 1	endian=big endian=little
3	__RENESAS_VERSION__ *1	.DEFINE *XXYYZZ*00H *2	-

	Macro Name	Value	Option
4	__RXV1	.DEFINE 1	isa=rxv1 *3
5	__RXV2	.DEFINE 1	isa=rxv2 *3
6	__RXV3 [V3.00.00 or later]	.DEFINE 1	isa=rxv3 *3
7	__RX_ISA_VERSION__ [V3.00.00 or later]	.DEFINE 1 .DEFINE 2 .DEFINE 3	isa=rxv1 *3 isa=rxv2 *3 isa=rxv3 *3
8	__ASRX__ *1 [V2.03.00 or later]	.DEFINE 1	-
9	__RENESAS__ *1 [V2.03.00 or later]	.DEFINE 1	-
10	__FPU	.DEFINE 1	-fpu
11	__DPFPU	.DEFINE 1	-dpfpu

Notes 1. Always defined regardless of the option.

Notes 2. When the Assembler version is V*XX*.*YY*.*ZZ*, the value of **__RENESAS_VERSION__** is *XXYYZZ*00H.
Example
For V3.01.00: **__RENESAS_VERSION__ .DEFINE 03010000H**.

Notes 3. Includes the specification by the ISA_RX environment variable.

5.5 Reserved Words

The assembler handles the same strings as assembler directives and mnemonics as reserved words. These reserved words have special functions and they cannot be used as label names or symbol names in assembly-language files. They are not case-sensitive; for example, "ABS" and "abs" are the same reserved word.

Reserved words are classified into the following types.

(1) Assembler directives
All assembler directives and all strings that begin with a period (.).

(2) Mnemonics
All mnemonics of the RX Family.

(3) Register and flag names
All register and flag names of the RX family.

(4) Operators
All operators described in this section.

(5) System labels
A system label is a name that begins with two periods and is generated by the assembler. All system labels are handled as reserved words.

6. SECTION SPECIFICATIONS

6.1 List of Section Names

This section describes the sections for CCRX.

Each of the regions for execution instructions and data of the relocatable files output by the assembler comprises a section. A section is the smallest unit for data placement in memory. Sections have the following properties.

- Section attributes
  code Stores execution instructions
  data Stores data that can be changed
  romdata Stores fixed data
- Format type
  Relative-address format:A section that can be relocated by the optimizing linkage editor.
  Absolute-address format:A section of which the address has been determined; it cannot be relocated by the optimizing linkage editor.
- Initial values
  Specifies whether there are initial values at the start of program execution. Data which has initial values and data which does not have initial values cannot be included in the same section. For example, if the initialization list for an array contains elements less than the number of elements for that array, elements without initial values specified in the initialization list are handled as if 0 is specified for the initial value.
- Write operations
  Specifies whether write operations are or are not possible during program execution.
- Boundary alignment number
  Values to correct the addresses of the sections. The optimizing linkage editor corrects addresses of the sections so that they are multiples of each of the boundary alignment numbers.

6.1.1 C/C++ Program Sections

The correspondence between memory areas and sections for C/C++ programs and the standard library is described in Table 6.1.

Table 6.1 Summary of Memory Area Types and Their Properties

No.	Name	Section		Format Type	Initial Value Write Operation	Alignment Number	Description
		Name	Attribute				
1	Program area	P *1*6	code	Relative	Yes Not possible	1 byte *7	Stores machine code
2	Constant area	C_8 *1 *2 *6 *8 *10	romdata	Relative	Yes Not possible	8 bytes	Stores **const** type data
		C *1*2*6*8	romdata	Relative	Yes Not possible	4 bytes	
		C_2 *1*2*6*8	romdata	Relative	Yes Not possible	2 bytes	
		C_1 *1*2*6*8	romdata	Relative	Yes Not possible	1 byte	

No.	Name	Section		Format Type	Initial Value	Align-ment Number	Description
		Name	Attribute		Write Opera-tion		
3	Initialized data area	D_8 *1 *2 *6 *8 *10	romdata	Relative	Yes Possible	8 bytes	Stores data with initial values
		D *1*2*6*8	romdata	Relative	Yes Possible	4 bytes	
		D_2 *1*2*6*8	romdata	Relative	Yes Possible	2 bytes	
		D_1 *1*2*6*8	romdata	Relative	Yes Possible	1 byte	
4	Uninitialized data area	B_8 *1 *2 *6 *8 *10	data	Relative	No Possible	8 bytes	Stores data without ini-tial values
		B *1*2*6*8	data	Relative	No Possible	4 bytes	
		B_2 *1*2*6*8	data	Relative	No Possible	2 bytes	
		B_1 *1*2*6*8	data	Relative	No Possible	1 byte	
5	**switch** state-ment branch table area	W *1*2	romdata	Relative	Yes Not Possible	4 bytes	Stores branch tables for **switch** statements
		W_2 *1*2	romdata	Relative	Yes Not Possible	2 bytes	
		W_1 *1*2	romdata	Relative	Yes Not Possible	1 byte	
6	C++ initial pro-cessing/ post-processing data area	C$INIT	romdata	Relative	Yes Not possible	4 bytes	Stores addresses of con-structors and destruc-tors called for global class objects
7	C++ virtual func-tion table area	C$VTBL	romdata	Relative	Yes Not possible	4 bytes	Stores data for calling the virtual function when a virtual function exists in the class declaration
8	User stack area	SU	data	Relative	No Possible	4 bytes	Area necessary for pro-gram execution
9	Interrupt stack area	SI	data	Relative	No Possible	4 bytes	Area necessary for pro-gram execution
10	Heap area	-	-	Relative	No Possible	-	Area used by library functions **malloc**, **real-loc**, **calloc**, **and new** *9
11	Absolute address vari-able area	$ADDR_ <section>_ <address> *3	data	Absolute	Yes/No Possible/ Not possible *4	-	Stores variables speci-fied by **#pragma address**

No.	Name	Section		Format Type	Initial Value Write Operation	Alignment Number	Description
		Name	Attribute				
12	Variable vector area	C$VECT C$VECT<vector table number>	romdata	Relative	No Possible	4 bytes	Variable vector table
13	Literal area	L *5	romdata	Relative	Yes Possible/ Not possible	4 bytes	Stores string literals and initializers used for dynamic initialization of aggregates

Notes 1. Section names can be switched using the **section** option.

Notes 2. Specifying a section with a boundary alignment of 4 when switching the section names also changes the section name of sections with a boundary alignment of 1, 2, or 8.

Notes 3. **<section>** is a **C**, **D**, or **B** section name, and **<address>** is an absolute address (hexadecimal).

Notes 4. The initial value and write operation depend on the attribute of **<section>**.

Notes 5. The section name can be changed by using the **section** option. In this case, the **C** section can be selected as the changed name.

Notes 6. The section name can be changed through **#pragma section**.

Notes 7. Specifying the **instalign4** or **instalign8** option, **#pragma instalign4,** or **#pragma instalign8** changes the boundary alignment to 4 or 8.

Notes 8. If an endian not matching the **endian** option has been specified in **#pragma endian**, a dedicated section is created to store the relevant data. At the end of the section name, **_B** is added for **#pragma endian big**, and **_L** is added for **#pragma endian little**.

Notes 9. Using these functions requires the allocation of at least 16 bytes of memory as a heap area.

Notes 10. This section is for storing double-precision floating-point data when the **dpfpu** option is specified.

Examples 1. A program example is used to demonstrate the correspondence between a C program and the compiler-generated sections.

C program

```
int a=1;
char b;
const short c=0;
void main(){
 ...
}
```

Areas generated by the compiler and stored data	Section name
`Program area (main(){...})`	`P`
`Constant area (c)`	`C_2`
`Initialized data area (a)`	`D`
`Uninitialized data area (b)`	`B_1`

Examples 2. A program example is used to demonstrate the correspondence between a C++ program and the compiler-generated sections.

C++ program

```
class A{
  int m;
  A(int p);
  ~A();
};
A a(1);
char b;
extern const char c='a';
short d=1;
void f(){...}
```

Areas generated by the compiler and stored data	Section name
Program area (f(){...})	P
Constant area (c)	C_1
Initialized data area (d)	D_2
Uninitialized data area (a) Uninitialized data area (b)	B B_1
Initial processing/ postprocessing data areas (&A::A, &A::~A)	C $INT

6.2 Assembly Program Sections

In assembly programs, the **.SECTION** control directive is used to begin sections and declare their attributes, and the **.ORG** control directive is used to declare the format types of sections.

For details on the control directives, refer to section 5.2, Directives.

Example An example of an assembly program section declaration is shown below.

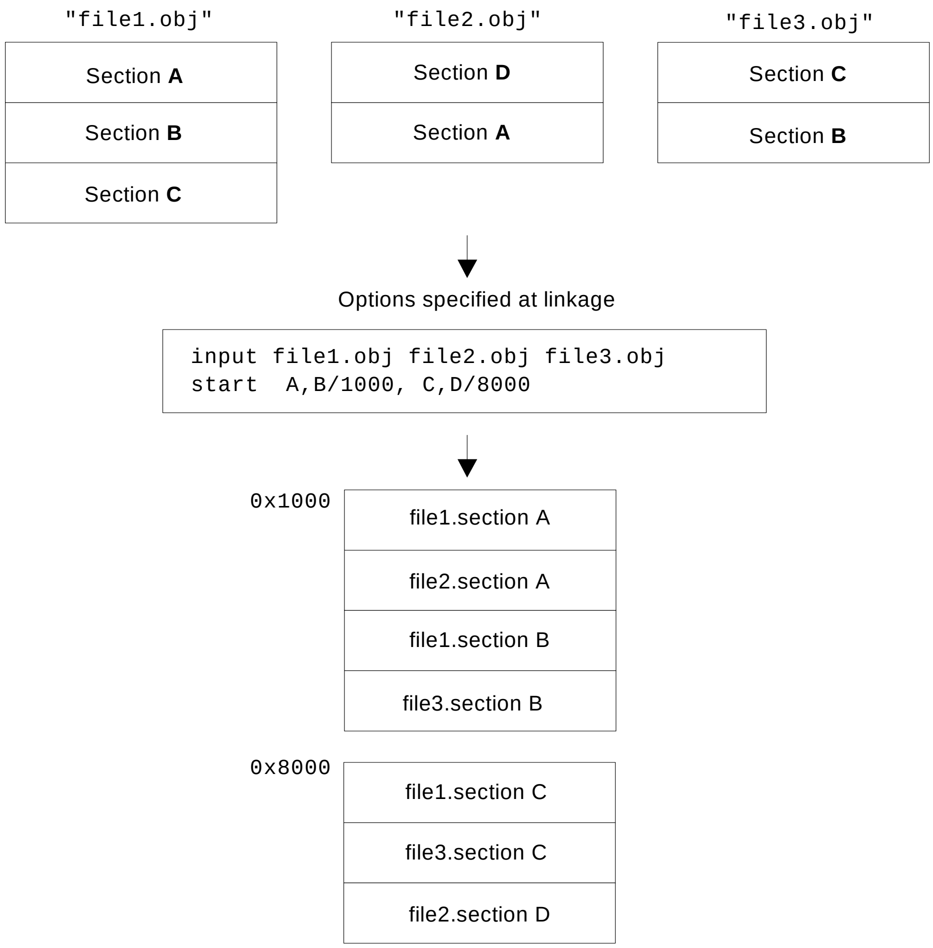
```
    .SECTION A,CODE,ALIGN=4  ;(1)

START:
    MOV.L  #CONST,R4
    MOV.L  [R4],R5
    ADD    #10,R5,R3
    MOV.L  #100,R4
    MOV.L  #ARRAY,R5
LOOP:
    MOV.L  R3,[R5+]
    SUB    #1,R4
    CMP    #0,R4
    BNE    LOOP
EXIT:
    RTS

;
    .SECTION B,ROMDATA  ;(2)
    .ORG   02000H
    .glb   CONST
CONST:
    .LWORD 05H
;
    .SECTION C,DATA,ALIGN=4  ;(3)
    .glb   BASE
BASE:
    .blkl  100
    .END
```

(1) Declares a **code** section with section name **A**, boundary alignment 4, and relative address format.
(2) Declares a **romdata** section with section name **B**, allocated address 2000H, and absolute address format.
(3) Declares a **data** section with section name **C**, boundary alignment 4, and relative address format.

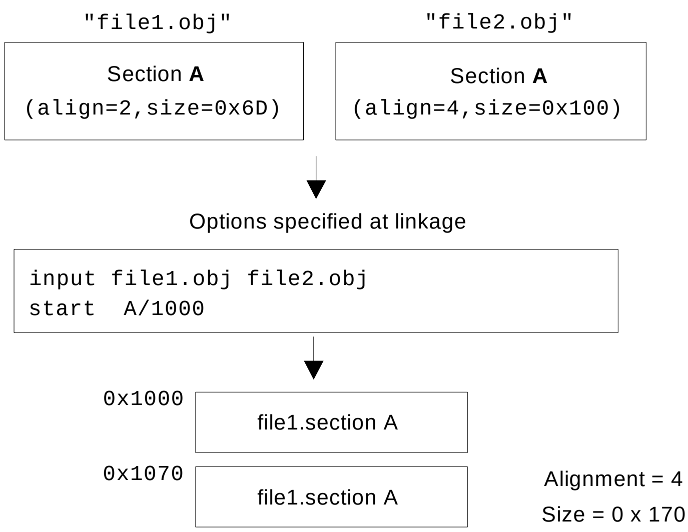
6.3 Linking Sections

The optimizing linkage editor links the same sections within input relocatable files, and allocates addresses specified by the **start** option.

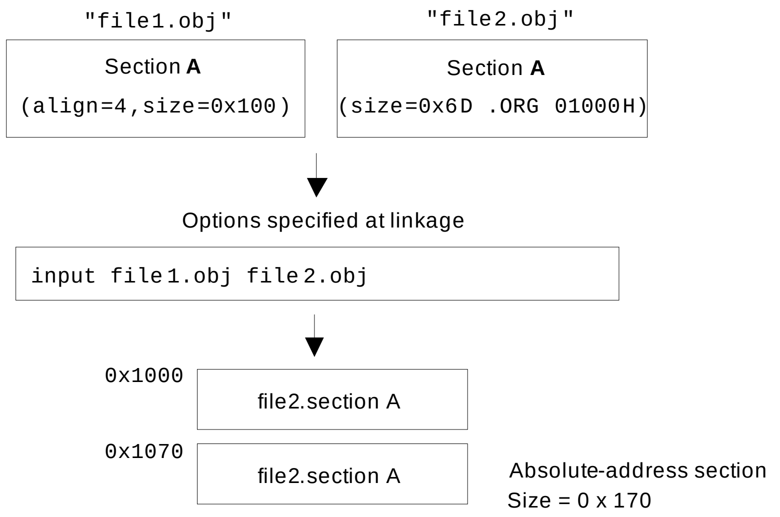
(1) The same section names in different files are allocated continuously in the order of file input.



(2) Sections with the same name but different boundary alignments are linked after alignment. Section alignment uses the larger of the section alignments.

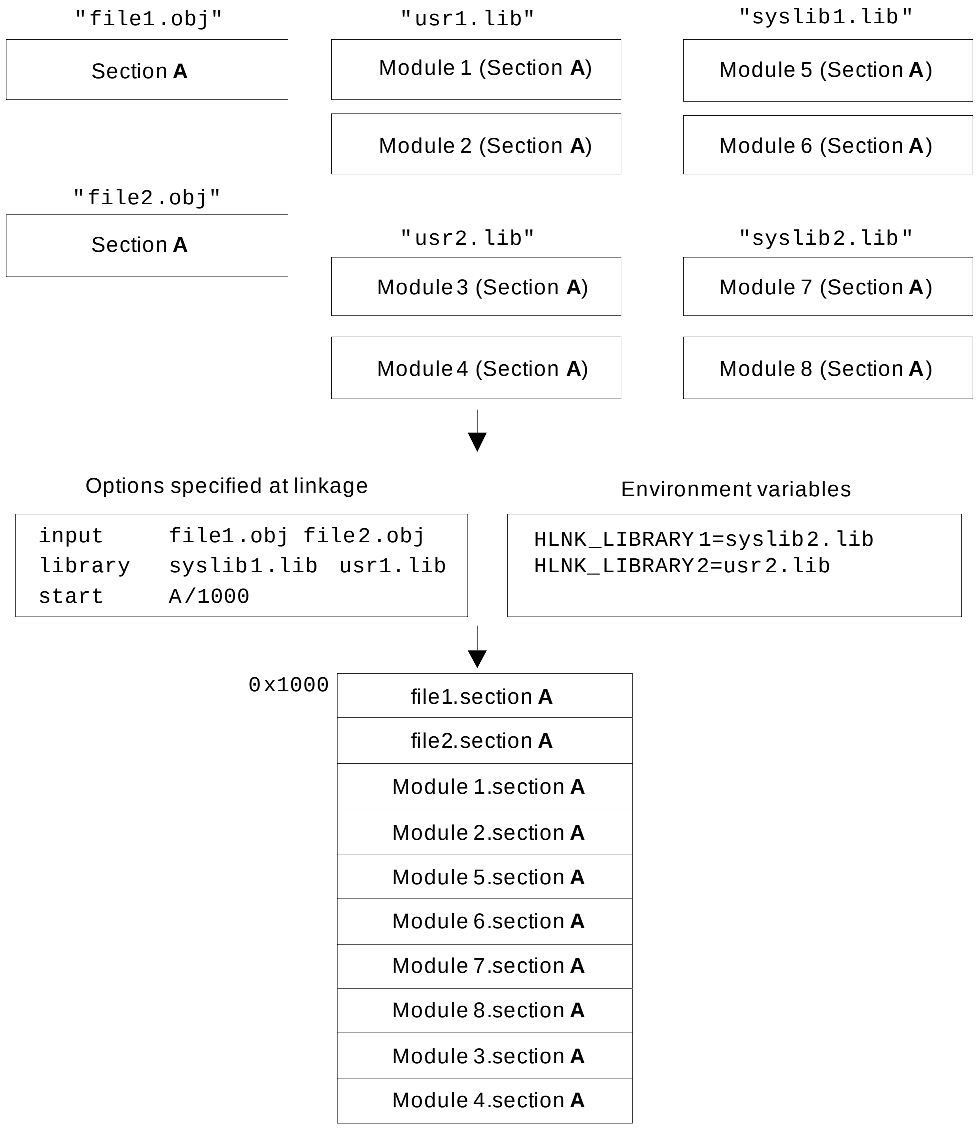


(3) When sections with the same name include both absolute-address and relative-address formats, relative-address sections are linked following absolute-address sections.



(4) Rules for the order of linking sections with the same name are based on their priorities as follows.

- Order specified by the **input** option or input files on the command line
- Order specified for the user library by the **library** option and order of input of modules within the library
- Order specified for the system library by the **library** option and order of input of modules within the library
- Order specified for libraries by environment variables (**HLNK_LIBRARY1** to **HLNK_LIBRARY3**) and order of input of modules within the library



7. LIBRARY FUNCTIONAL SPECIFICATION

This chapter provides library functions supplied with the CCRX.

7.1 Supplied Libraries

The CCRX provides the standard C, standard C99, and EC++ libraries.

7.1.1 Terms Used in Library Function Descriptions

(1) Stream input/output
In data input/output, it would lead to poor efficiency if each call of an input/output function, which handles a single character, drove the input/output device and the OS functions. To solve this problem, a storage area called a buffer is normally provided, and the data in the buffer is input or output at one time.
From the viewpoint of the program, on the other hand, it is more convenient to call input/output functions for each character.
Using the library functions, character-by-character input/output can be performed efficiently without awareness of the buffer status within the program by automatically performing buffer management.
Those library functions enable a programmer to write a program considering the input/output as a single data stream, making the programmer be able to implement data input/output efficiently without being aware of the detailed procedure. Such capability is called stream input/output.

(2) **FILE** structure and file pointer
The buffer and other information required for the stream input/output described above are stored in a single structure, defined by the name **FILE** in the **<stdio.h>** standard include file.
In stream input/output, all files are handled as having a **FILE** structure data structure. Files of this kind are called stream files. A pointer to this **FILE** structure is called a file pointer, and is used to specify an input/output file.
The file pointer is defined as
FILE *fp;
When a file is opened by the **fopen** function, etc., the file pointer is returned. If the open processing fails, **NULL** is returned. Note that if a **NULL** pointer is specified in another stream input/output function, that function will end abnormally. After opening a file, be sure to check the file pointer value to see whether the open processing has been successful.

(3) Functions and macros
There are two library function implementation methods: functions and macros.
A function has the same interface as an ordinary user-written function, and is incorporated during linkage. A macro is defined using a **#define** statement in the standard include file relating to the function.
The following points must be noted concerning macros:

- Macros are expanded automatically by the preprocessor, and therefore a macro expansion cannot be invalidated even if the user declares a function with the same name.
- If an expression with a side effect (assignment expression, increment, decrement) is specified as a macro parameter, its result is not guaranteed.

Example Macro definition of **MACRO** that calculates the absolute value of a parameter is as follows:
If the following definition is made:
```
#define  MACRO(a)  ((a) >= 0 ? (a) : -(a))
```
and if
X=MACRO(a++)
is in the program, the macro will be expanded as follows:
X = ((a++) >= 0 ? (a++) : -(a++))
a will be incremented twice, and the resultant value will be different from the absolute value of the initial value of a.

(4) **EOF**
In functions such as **getc**, **getchar**, and **fgetc**, which input data from a file, **EOF** is the value returned at end-of-file. The name **EOF** is defined in the **<stdio.h>** standard include file.

(5) **NULL**
This is the value indicating that a pointer is not pointing at anything. The name **NULL** is defined in the **<stddef.h>** standard include file.

(6) Null character
The end of a string in C/C++ is indicated by the characters \0. String parameters in library functions must also conform to this convention. The characters \0 indicating the end of a string are called null characters.

(7) Return code
With some library functions, a return value is used to determine the result (such as whether the specified processing succeeded or failed). In this case, the return value is called the return code.

(8) Text files and binary files
Many systems have special file formats to store data. To support this facility, library functions have two file formats: text files and binary files.

- Text files
A text file is used to store ordinary text, and consists of a collection of lines. In text file input, the new-line character (\n) is input as a line separator. In output, output of the current line is terminated by outputting the new-line character (\n). Text files are used to input/output files that store standard text for each system. With text files, characters input or output by a library function do not necessarily correspond to a physical stream of data in the file.

- Binary files
A binary file is configured as a row of byte data. Data input or output by a library function corresponds to a physical list of data in the file.

(9) Standard input/output files
Files that can be used as standard by input/output library functions by default without preparations such as opening file are called standard input/output files. Standard input/output files comprise the standard input file (**stdin**), standard output file (**stdout**), and standard error output file (**stderr**).

- Standard input file (**stdin**)
Standard file to be input to a program.

- Standard output file (**stdout**)
Standard file to be output from a program.

- Standard error output file (**stderr**)
Standard file for storing output of error messages, etc., from a program.

(10) Floating-point numbers
Floating-point numbers are numbers represented by approximation of real numbers. In a C source program, floating-point numbers are represented by decimal numbers, but inside the computer they are normally represented by binary numbers.
In the case of binary numbers, the floating-point representation is as follows:
$2^n \times m$ (n: integer, m: binary fraction)
Here, **n** is called the exponent of the floating-point number, and **m** is called the mantissa. The numbers of bits to represent **n** and **m** are normally fixed so that a floating-point number can be represented using a specific data size.
Some terms relating to floating-point numbers are explained below.

- Radix
An integer value indicating the number of distinct digits in the number system used by a floating-point number (10 for decimal, 2 for binary, etc.). The radix is normally 2.

- Rounding
Rounding is performed when an intermediate result of an operation of higher precision than a floating-point type is stored as that floating-point type. There is rounding up, rounding down, and half-adjust rounding (i.e., in binary representation, rounding down 0 and rounding up 1).

- Normalization
When a floating-point number is represented in the form $2^n \times m$, the same number can be represented in different ways.
[Format] The following two expressions represent the same value.
$2^5 \times 1.0_{(2)}$ ($_{(2)}$ indicates a binary number)
$2^6 \times 0.1_{(2)}$
Usually, a representation in which the leading digit is not 0 is used, in order to secure the number of valid digits. This is called a normalized floating-point number, and the operation that converts a floating-point number to this kind of representation is called normalization.

- Guard bit
  When saving an intermediate result of a floating-point operation, data one bit longer than the actual floating-point number is normally provided in order for rounding to be carried out. However, this alone does not permit an accurate result to be achieved in the event of digit dropping, etc. For this reason, the intermediate result is saved with an extra bit, called a guard bit.

(11) File access mode
This is a string that indicates the kind of processing to be carried out on a file when it is opened. There are 12 different modes, as shown in Table 7.1.

Table 7.1 File Access Modes

Access Mode	Meaning
'r'	Opens text file for reading
'w'	Opens text file for writing
'a'	Opens text file for addition
'rb'	Opens binary file for reading
'wb'	Opens binary file for writing
'ab'	Opens binary file for appending
'r+'	Opens text file for reading and updating
'w+'	Opens text file for writing and updating
'a+'	Opens text file for appending and updating
'r+b'	Opens binary file for reading and updating
'w+b'	Opens binary file for writing and updating
'a+b'	Opens binary file for appending and updating

(12) implementation defined
Definitions differ for each compiler.

(13) Error indicator and end-of-file indicator
The following two data items are held for each stream file: (1) an error indicator that indicates whether or not an error has occurred during file input/output, and (2) an end-of-file indicator that indicates whether or not the input file has ended.
These data items can be referenced by the **ferror** function and the **feof** function, respectively.
With some functions that handle stream files, error occurrence and end-of-file information cannot be obtained from the return value alone. The error indicator and end-of-file indicator are useful for checking the file status after execution of such functions.

(14) File position indicator
Stream files that can be read or written at any position within the file, such as disk files, have an associated data item called a file position indicator that indicates the current read/write position within the file.
File position indicators are not used with stream files that do not permit the read/write position within the file to be changed, such as terminals.

7.1.2 Notes on Use of Libraries

The contents of macros defined in a library differ for each compiler.
When a library is used, the behavior is not guaranteed if the contents of these macros are redefined.
With libraries, errors are not detected in all cases. The behavior is not guaranteed if library functions are called in a form other than those shown in the descriptions in the following sections.

7.2 Header Files

The list of header files required for using the libraries of the RX are listed below.
The macro definitions and function declarations are described in each file.

Table 7.2 Library Types and Corresponding Standard Include Files

Library Type	Description	Standard Include File
Program diagnostics	Outputs program diagnostic information.	<assert.h>
Character handling	Handles and checks characters.	<ctype.h>
Mathematics	Performs numerical calculations such as trigonometric functions.	<math.h> <mathf.h>
Non-local jumps	Supports transfer of control between functions.	<setjmp.h>
Variable arguments	Supports access to variable arguments for functions with such arguments.	<stdarg.h>
Input/output	Performs input/output handling.	<stdio.h>
General utilities	Performs C program standard processing such as storage area management.	<stdlib.h>
String handling	Performs string comparison, copying, etc.	<string.h>
Complex arithmetic	Performs complex number operations.	<complex.h>
Floating-point environment	Supports access to floating-point environment.	<fenv.h>
Integer type format conversion	Manipulates greatest-width integers and converts integer format.	<inttypes.h>
Multibyte and wide characters	Manipulates multibyte characters.	<wchar.h> <wctype.h>

In addition to the above standard include files, standard include files consisting solely of macro name definitions, shown in Table 7.3, are provided to improve programming efficiency.

Table 7.3 Standard Include Files Comprising Macro Name Definitions

Standard Include File	Description
<stddef.h>	Defines macro names used in common by the standard include files.
<limits.h>	Defines various limit values relating to compiler internal processing.
<errno.h>	Defines the value to be set in **errno** when an error is generated in a library function.
<float.h>	Defines various limit values relating to the limits of floating-point numbers.
<iso646.h>	Defines alternative spellings of macro names.
<stdbool.h>	Defines macros relating to logical types and values.
<stdint.h>	Declares integer types with specified width and defines macros.
<tgmath.h>	Defines type-generic macros.

7.3 Reentrant Library

Functions of libraries created by using the standard library generator with the **-reent** option specified can be executed as reentrant except for the rand and srand functions and the functions of the **EC++** library.

Table 7.4 and Table 7.5 indicate which functions of libraries are reentrant even when the **-reent** option is not specified.

In the tables, functions for which "**D**" is indicated set the **errno** variable, so execution as reentrant is only possible as long as the program does not refer to the **errno** variable.

Reentrant column O: Reentrant X: Non-reentrant D: Sets the errno variable.

Table 7.4 C(C89) Reentrant Library Function List

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
stddef.h	offsetof	O	math.h	frexp	D
assert.h	assert	X		ldexp	D
ctype.h	isalnum	O		log	D
	isalpha	O		log10	D
	iscntrl	O		modf	D
	isdigit	O		pow	D
	isgraph	O		sqrt	D *1
	islower	O		ceil	D
	isprint	O		fabs	D *1
	ispunct	O		floor	D
	isspace	O		fmod	D
	isupper	O	mathf.h	acosf	D *2
	isxdigit	O		asinf	D *2
	tolower	O		atanf	D *2
	toupper	O		atan2f	D *2
math.h	acos	D *2		cosf	D *2
	asin	D *2		sinf	D *2
	atan	D *2		tanf	D *2
	atan2	D *2		coshf	D
	cos	D *2		sinhf	D
	sin	D *2		tanhf	D
	tan	D *2		expf	D
	cosh	D		frexpf	D
	sinh	D		ldexpf	D
	tanh	D		logf	D
	exp	D		log10f	D

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
mathf.h	modff	D	stdio.h	fputs	X
	powf	D		getc	X
	sqrtf	D *1		getchar	X
	ceilf	D		gets	X
	fabsf	O		putc	X
	fabsl	D *1		putchar	X
	floorf	D		puts	X
	fmodf	D		ungetc	X
setjmp.h	setjmp	O		fread	X
	longjmp	O		fwrite	X
stdarg.h	va_start	O		fseek	X
	va_arg	O		ftell	X
	va_end	O		rewind	X
stdio.h	fclose	X		clearerr	X
	fflush	X		feof	X
	fopen	X		ferror	X
	freopen	X		perror	X
	setbuf	X	stdlib.h	atof	D
	setvbuf	X		atoi	D
	fprintf	X		atol	D
	fscanf	X		atoll	D
	printf	X		strtod	D
	scanf	X		strtol	D
	sprintf	X		strtoul	D
	sscanf	D		strtoll	D
	vfprintf	X		strtoull	D
	vprintf	X		rand	X
	vsprintf	X		srand	X
	fgetc	X		calloc	X
	fgets	X		free	X
	fputc	X		malloc	X

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
stdlib.h	realloc	X	string.h	strcmp	O
	bsearch	O		strncmp	O
	qsort	O		memchr	O
	abs	O		strchr	O
	div	O		strcspn	O
	labs	O		strpbrk	O
	llabs	O		strrchr	O
	ldiv	O		strspn	O
	lldiv	O		strstr	O
string.h	memcpy	O		strtok	X
	strcpy	O		memset	O
	strncpy	O		strerror	O
	strcat	O		strlen	O
	strncat	O		memmove	O
	memcmp	O			

Notes 1. If the function call is replaced by an instruction, the entry in the column for "Reentrant" in the table would become O (i.e. reentrance is possible) since the instruction does not update the **errno** variable. Refer to the item on **-library** in the section on compiler options for the conditions under which calls are replaced by instructions.

Notes 2. If the function call is replaced by code that uses a trigonometric function unit, the entry in the column for "Reentrant" in the table might become X (i.e. non-reentrant). For details, refer to the item about **-tfu** in the section on compile options.

Table 7.5 C99 Reentrant Library Functions List

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
stddef.h	isblank	O	math.h	frexpl	D
math.h	acosl	D *2		ldexpl	D
	asinl	D *2		logl	D
	atanl	D *2		log10l	D
	atan2l	D *2		modfl	D
	cosl	D *2		powl	D
	sinl	D *2		sqrtl	D *1
	tanl	D *2		ceill	D
	coshl	D		fabsl	D *1
	sinhl	D		floorl	D
	tanhl	D		fmodl	D
	expl	D		fpclassify	O

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
math.h	isfinite	O	math.h	log2f	X
	isinf	O		log2l	X
	isnan	O		logb	X
	isnormal	O		logbf	X
	signbit	O		logbl	X
	isgreater	O		scalbn	X
	isgreaterequal	O		scalbnf	X
	isless	O		scalbnl	X
	islessequal	O		scalbln	X
	islessgreater	O		scalblnf	X
	isunordered	O		scalblnl	X
	acosh	X		cbrt	O
	acoshf	X		cbrtf	O
	acoshl	X		cbrtl	O
	asinh	X		hypot	X
	asinhf	X		hypotf	X
	asinhl	X		hypotl	X
	atanh	X		erf	X
	atanhf	X		erff	X
	atanhl	X		erfl	X
	exp2	X		erfc	X
	exp2f	X		erfcf	X
	exp2l	X		erfcl	X
	expm1	D		lgamma	X
	expm1f	D		lgammaf	X
	expm1l	D		lgammal	X
	ilogb	O		tgamma	X
	ilogbf	O		tgammaf	X
	ilogbl	O		tgammal	X
	log1p	X		nearbyint	O
	log1pf	X		nearbyintf	O
	log1pl	X		nearbyintl	O
	log2	X		rint	X

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
math.h	rintf	X	math.h	nextafterf	X
	rintl	X		nextafterl	X
	lrint	X		nexttoward	X
	lrintf	X		nexttowardf	X
	lrintl	X		nexttowardl	X
	llrint	X		fdim	O
	llrintf	X		fdimf	O
	llrintl	X		fdiml	O
	round	O		fmax	O
	roundf	O		fmaxf	O
	roundl	O		fmaxl	O
	lround	X		fmin	O
	lroundf	X		fminf	O
	lroundl	X		fminl	O
	llround	X		fma	X
	llroundf	X		fmaf	X
	llroundl	X		fmal	X
	trunc	O	stdarg.h	va_copy	O
	truncf	O	stdio.h	snprintf	X
	truncl	O		vsnprintf	X
	remainder	X		vfscanf	X
	remainderf	X		vscanf	X
	remainderl	X		vsscanf	D
	remquo	X	complex.h	cacos	X
	remquof	X		cacosf	X
	remquol	X		cacosl	X
	copysign	O		casin	X
	copysignf	O		casinf	X
	copysignl	O		casinl	X
	nan	O		catan	X
	nanf	O		catanf	X
	nanl	O		catanl	X
	nextafter	X		ccos	X

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
complex.h	ccosf	X	complex.h	cabsl	X
	ccosl	X		cpow	X
	csin	X		cpowf	X
	csinf	X		cpowl	X
	csinl	X		csqrt	D
	ctan	D		csqrtf	D
	ctanf	D		csqrtl	D
	ctanl	D		carg	D
	cacosh	X		cargf	D
	cacoshf	X		cargl	D
	cacoshl	X		cimag	O
	casinh	X		cimagf	O
	casinhf	X		cimagl	O
	casinhl	X		conj	O
	catanh	X		conjf	O
	catanhf	X		conjl	O
	catanhl	X		cproj	O
	ccosh	X		cprojf	O
	ccoshf	X		cprojl	O
	ccoshl	X		creal	O
	csinh	X		crealf	O
	csinhf	X		creall	O
	csinhl	X	fenv.h	feclearexcept	X
	ctanh	D		fegetexceptflag	O
	ctanhf	D		feraiseexcept	X
	ctanhl	D		fesetexceptflag	X
	cexp	X		fetestexcept	O
	cexpf	X		fegetround	O
	cexpl	X		fesetround	X
	clog	X		fegetenv	O
	clogf	X		feholdexcept	X
	clogl	X		fesetenv	X
	cabs	X		feupdateenv	X
	cabsf	X			

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
inttypes.h	imaxabs	O	wchar.h	wcstod	D
	imaxdiv	O		wcstof	D
	strtoimax	D		wcstold	D
	strtoumax	D		wcstol	D
	wcstoimax	D		wcstoll	D
	wcstoumax	D		wcstoul	D
wchar.h	fwprintf	X		wcstoull	D
	vfwprintf	X		wcscpy	O
	swprintf	X		wcsncpy	O
	vswprintf	X		wmemcpy	O
	wprintf	X		wmemmove	O
	vwprintf	X		wcscat	O
	fwscanf	X		wcsncat	O
	vfwscanf	X		wcscmp	O
	swscanf	D		wcsncmp	O
	vswscanf	D		wmemcmp	O
	wscanf	X		wcschr	O
	vwscanf	X		wcscspn	O
	fgetwc	X		wcspbrk	O
	fgetws	X		wcsrchr	O
	fputwc	X		wcsspn	O
	fputws	X		wcsstr	O
	fwide	X		wcstok	O
	getwc	X		wmemchr	O
	getwchar	X		wcslen	O
	putwc	X		wmemset	O
	putwchar	X		mbsinit	O
	ungetwc	X		mbrlen	X

Notes 1. If the function call is replaced by an instruction, the entry in the column for "Reentrant" in the table would become O (i.e. reentrance is possible) since the instruction does not update the **errno** variable. Refer to the item on **-library** in the section on compiler options for the conditions under which calls are replaced by instructions.

Notes 2. If the function call is replaced by code that uses a trigonometric function unit, the entry in the column for "Reentrant" in the table might become X (i.e. non-reentrant). For details, refer to the item about **-tfu** in the section on compile options.

7.4 Library Function

This section explains library functions.

Some of the C99-language-expanded keywords (functions, macros, variable names, etc..) must be used in when the C99-language is selected. Such keywords are displayed by the mark of "<-*lang*=*c99*>" at the tables each the header-files in these sections. When you use these keywords in your program, at the time of compilations and library generations, please turn on the -lang=c99 option.

7.4.1 <stddef.h>

Defines macro names used in common in the standard include files.
The following macro names are all implementation-defined.

Type	Definition Name	Description
Type (typedef)	ptrdiff_t	Indicates the type of the result of subtraction between two pointers.
	size_t	Indicates the type of the result of an operation using the **sizeof** operator.
Constant (macro)	NULL	Indicates the value when a pointer is not pointing at anything. This value is such that the result of a comparison with 0 using the equality operator (==) is true.
Variable (macro)	errno	If an error occurs during library function processing, the error code defined in the respective library is set in **errno**. By setting 0 in **errno** before calling a library function and checking the error code set in **errno** after the library function processing has ended, it is possible to check whether an error occurred during the library function processing.
Function (macro)	offsetof	Obtains the offset in bytes from the beginning of a structure to a structure member.
Type (typedef)	wchar_t	Type that indicates an extended character.

Implementation-Defined Specifications

Item	Compiler Specifications
Value of macro **NULL**	Value 0.
Type equivalent to macro **ptrdiff_t**	**long** type
Type equivalent to **wchar_t**	**unsigned short** type

7.4.2 <assert.h>

Adds diagnostics into programs.

Type	Definition Name	Description
Function (macro)	assert	Adds diagnostics into programs.

To invalidate the diagnostics defined by **<assert.h>**, define macro name **NDEBUG** with a **#define** statement (**#define NDEBUG**) before including **<assert.h>**.

Note If an **#undef** statement is used for macro name **assert**, the result of subsequent **assert** calls is not guaranteed.

assert

Adds diagnostics into programs.

[Format]

```
#include <assert.h>
void assert (long expression);
```

[Parameters]

expression Expression to be evaluated.

[Remarks]

When **expression** is true, the **assert** macro terminates processing without returning a value. If **expression** is false, it outputs diagnostic information to the standard error file in the form defined by the compiler, and then calls the **abort** function.

The diagnostic information includes the parameter's program text, source file name, and source line numbers.

Implementation define:

The following message is output when **expression** is false in **assert (expression)**:

The message depends on the **lang** option setting at compilation.

(1) When **-lang=c99** is not specified (C (C89), C++, or EC++ language):

```
ASSERTION FAILED:expressionFILE<file name>,LINE<line number>
```

(2) When **-lang=c99** is specified (C (C99) language):

```
ASSERTION FAILED:expressionFILE<file name>,LINE<line number>FUNCNAME<function name>
```

7.4.3 <ctype.h>

Checks and converts character types.

Type	Definition Name	Description
Function	isalnum	Tests for a letter or a decimal digit.
	isalpha	Tests for a letter.
	iscntrl	Tests for a control character.
	isdigit	Tests for a decimal digit.
	isgraph	Tests for a printing character except space.
	islower	Tests for a lowercase letter.
	isprint	Tests for a printing character including space.
	ispunct	Tests for a special character.
	isspace	Tests for a white-space character.
	isupper	Tests for an uppercase letter.
	isxdigit	Tests for a hexadecimal digit.
	tolower	Converts an uppercase letter to lowercase.
	toupper	Converts a lowercase letter to uppercase.
	isblank *<-lang=c99>*	Tests for a space character or a tab character.

In the above functions, if the input parameter value is not within the range that can be represented by the **unsigned char** type and is not **EOF**, the operation of the function is not guaranteed.

Character types are listed in Table 7.6.

Table 7.6 Character Types

Character Type	Description
Uppercase letter	Any of the following 26 characters 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
Lowercase letter	Any of the following 26 characters 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'
Letter	Any uppercase or lowercase letter
Decimal digit	Any of the following 10 characters '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
Printing character	A character, including space (' ') that is displayed on the screen (corresponding to ASCII codes 0x20 to 0x7E)
Control character	Any character except a printing character
White-space character	Any of the following 6 characters Space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), vertical tab ('\v')
Hexadecimal digit	Any of the following 22 characters '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'a', 'b', 'c', 'd', 'e', 'f'
Special character	Any printing character except space (' '), a letter, or a decimal digit

Character Type	Description
Blank character	Either of the following 2 characters Space (' '), horizontal tab ('\t')

Implementation-Defined Specifications

Item	Compiler Specifications
The character set inspected by the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions	Character set represented by the **unsigned char** type (0 to 255) and **EOF** (-1). Table 7.7 shows the character set that results in a true return value.

Table 7.7 True Character

Function Name	True Characters
isalnum	'0' to '9', 'A' to 'Z', 'a' to 'z'
isalpha	'A' to 'Z', 'a' to 'z'
iscntrl	'\x00' to '\x1f', '\x7f'
islower	'a' to 'z'
isprint	'\x20' to '\x7E'
isupper	'A' to 'Z'

isalnum

Tests for a letter or a decimal digit.

[Format]

```
#include <ctype.h>
long isalnum (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is a letter or a decimal digit: Nonzero
If character **c** is not a letter or a decimal digit: 0

isalpha

Tests for a letter.

[Format]

```
#include <ctype.h>
long isalpha(long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is a letter: Nonzero
If character **c** is not a letter: 0

iscntrl

Tests for a control character.

[Format]

```
#include <ctype.h>
long iscntrl (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is a control character: Nonzero
If character **c** is not a control character: 0

isdigit

Tests for a decimal digit.

[Format]

```
#include <ctype.h>
long isdigit (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character c is a decimal digit: Nonzero
If character c is not a decimal digit: 0

isgraph

Tests for any printing character except space (' ').

[Format]

```
#include <ctype.h>
long isgraph (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is a printing character except space: Nonzero
If character **c** is not a printing character except space: 0

islower

Tests for a lowercase letter.

[Format]

```
#include <ctype.h>
long islower (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is a lowercase letter: Nonzero
If character **c** is not a lowercase letter: 0

isprint

Tests for a printing character including space (' ').

[Format]

```
#include <ctype.h>
long isprint (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is a printing character including space: Nonzero
If character **c** is not a printing character including space: 0

ispunct

Tests for a special character.

[Format]

```
#include <ctype.h>
long ispunct (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is a special character: Nonzero
If character **c** is not a special character: 0

isspace

Tests for a white-space character.

[Format]

```
#include <ctype.h>
long isspace (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is a white-space character: Nonzero
If character **c** is not a white-space character: 0

isupper

Tests for an uppercase letter.

[Format]

```
#include <ctype.h>
long isupper (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is an uppercase letter: Nonzero
If character **c** is not an uppercase letter: 0

isxdigit

Tests for a hexadecimal digit.

[Format]

```
#include <ctype.h>
long isxdigit (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character **c** is a hexadecimal digit: Nonzero
If character **c** is not a hexadecimal digit: 0

tolower

Converts an uppercase letter to the corresponding lowercase letter.

[Format]

```
#include <ctype.h>
long tolower (long c);
```

[Parameters]

c Character to be converted

[Return values]

If character **c** is an uppercase letter:Lowercase letter corresponding to character **c**
If character **c** is not an uppercase letter: Character **c**

toupper

Converts a lowercase letter to the corresponding uppercase letter.

[Format]

```
#include <ctype.h>
long toupper (long c);
```

[Parameters]

c Character to be converted

[Return values]

If character **c** is a lowercase letter: Uppercase letter corresponding to character **c**
If character **c** is not a lowercase letter: Character **c**

isblank

Tests for a space character or a tab character.

[Format]

```
#include <ctype.h>
long isblank (long c);
```

[Parameters]

c Character to be tested

[Return values]

If character c is a space character or a tab character: Nonzero
If character c is neither a space character nor a tab character: 0

7.4.4 <float.h>

Defines various limits relating to the internal representation of floating-point numbers.
The following macro names are all implementation-defined.

Type	Definition Name	Definition Value	Description
Constant (macro)	FLT_RADIX	2	Indicates the radix in exponent representation.
	FLT_ROUNDS	1	Indicates whether or not the results of addition are rounded off. 1 means that rounding of the results of operation is enabled.
	FLT_MAX	3.4028235677973364e+38F	Indicates the maximum value that can be represented as a **float** type floating-point value.
	DBL_MAX	1.7976931348623158e+308	Indicates the maximum value that can be represented as a **double** type floating-point value.
	LDBL_MAX	1.7976931348623158e+308	Indicates the maximum value that can be represented as a **long double** type floating-point value.
	FLT_MAX_EXP	128	Using powers of the radix two to represent the range of **float** type floating-point numbers, FLT_MAX_EXP indicates the maximum value of the exponent plus 1.
	DBL_MAX_EXP	1024	Using powers of the radix two to represent the range of **double** type floating-point numbers, DBL_MAX_EXP indicates the maximum value of the exponent plus 1.
	LDBL_MAX_EXP	1024	Using powers of the radix two to represent the range of **long double** type floating-point numbers, LDBL_MAX_EXP indicates the maximum value of the exponent plus 1.
	FLT_MAX_10_EXP	38	Using powers of 10 to represent the range of **float** type floating point positive numbers, FLT_MAX_10_EXP indicates the exponent of the maximum integer.
	DBL_MAX_10_EXP	308	Using powers of 10 to represent the range of **double** type floating point positive numbers, DBL_MAX_10_EXP indicates the exponent of the maximum integer.
	LDBL_MAX_10_EXP	308	Using powers of 10 to represent the range of **long double** type floating point positive numbers, LDBL_MAX_10_EXP indicates the exponent of the maximum integer.
	FLT_MIN	1.175494351e–38F	Indicates the minimum positive value that can be represented as a **float** type floating-point value.
	DBL_MIN	2.2250738585072014e–308	Indicates the minimum positive value that can be represented as a **double** type floating-point value.
	LDBL_MIN	2.2250738585072014e–308	Indicates the minimum positive value that can be represented as a **long double** type floating-point value.

Type	Definition Name	Definition Value	Description
Constant (macro)	FLT_MIN_EXP	–125	Using powers of the radix two to represent the range of **float** type floating-point numbers, FLT_MIN_EXP indicates the minimum value of the exponent plus 1.
	DBL_MIN_EXP	–1021	Using powers of the radix two to represent the range of **double** type floating-point numbers, DBL_MIN_EXP indicates the minimum value of the exponent plus 1.
	LDBL_MIN_EXP	–1021	Using powers of the radix two to represent the range of **long double** type floating-point numbers, LDBL_MIN_EXP indicates the minimum value of the exponent plus 1.
	FLT_MIN_10_EXP	–37	Using powers of 10 to represent the range of **float** type floating point positive numbers, FLT_MIN_10_EXP indicates the exponent of the minimum integer.
	DBL_MIN_10_EXP	–307	Using powers of 10 to represent the range of **double** type floating point positive numbers, DBL_MIN_10_EXP indicates the exponent of the minimum integer.
	LDBL_MIN_10_EXP	–307	Using powers of 10 to represent the range of **long double** type floating point positive numbers, LDBL_MIN_10_EXP indicates the exponent of the minimum integer.
	FLT_DIG	6	Indicates the maximum number of digits in **float** type floating-point value decimal-precision.
	DBL_DIG	15	Indicates the maximum number of digits in **double** type floating-point value decimal-precision.
	LDBL_DIG	15	Indicates the maximum number of digits in **long double** type floating-point value decimal-precision.
	FLT_MANT_DIG	24	Indicates the maximum number of mantissa digits when a **float** type floating-point value is represented in the radix.
	DBL_MANT_DIG	53	Indicates the maximum number of mantissa digits when a **double** type floating-point value is represented in the radix.
	LDBL_MANT_DIG	53	Indicates the maximum number of mantissa digits when a **long double** type floating-point value is represented in the radix.
	DECIMAL_DIG	17	Indicates the maximum number of digits of a floating-point value represented in decimal precision.
	FLT_EPSILON	1.1920928955078125e-07	Indicates the difference between 1 and the minimum value greater than 1 that can be represented in **float** type.

Type	Definition Name	Definition Value	Description
Constant (macro)	DBL_EPSILON	2.2204460492503131e-16	Indicates the difference between 1 and the minimum value greater than 1 that can be represented in **double** type.
	LDBL_EPSILON	2.2204460492503131e-16	Indicates the difference between 1 and the minimum value greater than 1 that can be represented in **long double** type.

7.4.5 <limits.h>

Defines various limits relating to the internal representation of integer type data.
The following macro names are all implementation-defined.

Type	Definition Name	Definition Value	Description
Constant (macro)	CHAR_BIT	8	Indicates the number of bits in a **char** type value.
	CHAR_MAX	127 255[*1]	Indicates the maximum value that can be represented by a **char** type variable.
	CHAR_MIN	–128 0[*1]	Indicates the minimum value that can be represented by a **char** type variable.
	SCHAR_MAX	127	Indicates the maximum value that can be represented by a **signed char** type variable.
	SCHAR_MIN	–128	Indicates the minimum value that can be represented by a **signed char** type variable.
	UCHAR_MAX	255U	Indicates the maximum value that can be represented by an **unsigned char** type variable.
	SHRT_MAX	32767	Indicates the maximum value that can be represented by a **short** type variable.
	SHRT_MIN	–32768	Indicates the maximum value that can be represented by a **short** type variable.
	USHRT_MAX	65535U	Indicates the maximum value that can be represented by an **unsigned short** type variable.
	INT_MAX	217483647	Indicates the maximum value that can be represented by an **int** type variable.
	INT_MIN	–2147483647–1	Indicates the minimum value that can be represented by an **int** type variable.
	UINT_MAX	4294967295U	Indicates the maximum value that can be represented by an **unsigned int** type variable.
	LONG_MAX	217483647L	Indicates the maximum value that can be represented by a **long** type variable.
	LONG_MIN	–2147483647L–1L	Indicates the minimum value that can be represented by a **long** type variable.
	ULONG_MAX	4294967295U	Indicates the maximum value that can be represented by an **unsigned long** type variable.
	LLONG_MAX	9223372036854775807LL	Indicates the maximum value that can be represented by a **long long** type variable.
	LLONG_MIN	–9223372036854775807LL –1LL	Indicates the minimum value that can be represented by a **long long** type variable.
	ULLONG_MAX	18446744073709551615ULL	Indicates the maximum value that can be represented by an **unsigned long long** type variable.

Notes 1. Indicates the value that can be represented by a variable when the **signed_char** option is specified.

7.4.6 <errno.h>

Defines the value to be set in **errno** when an error is generated in a library function.
The following macro names are all implementation-defined.

Type	Definition Name	Description
Variable (macro)	errno	**int** type variable. An error number is set when an error is generated in a library function.
Constant (macro)	ERANGE	Refer to section 10.5.6, Standard Library Error Messages.
	EDOM	
	ESTRN	
	PTRERR	
	ECBASE	
	ETLN	
	EEXP	
	EEXPN	
	EFLOATO	
	EFLOATU	
	EDBLO	
	EDBLU	
	ELDBLO	
	ELDBLU	
	NOTOPN	
	EBADF	
	ECSPEC	
	EFIXEDO	
	EFIXEDU	
	EACCUMO	
	EACCUMU	
	EILSEQ	

7.4.7 <math.h>

Performs various mathematical operations.
The following constants (macros) are all implementation-defined.

Type	Definition Name	Description
Constant (macro)	EDOM	Indicates the value to be set in **errno** if the value of a parameter input to a function is outside the range of values defined in the function.
	ERANGE	Indicates the value to be set in **errno** if the result of a function cannot be represented as a **double** type value, or if an overflow or an underflow occurs.
	HUGE_VAL HUGE_VALF *<-lang=c99>* HUGE_VALL *<-lang=c99>*	Indicates the value for the function return value if the result of a function overflows.
	INFINITY *<-lang=c99>*	Expanded to a **float**-type constant expression that represents positive or unsigned infinity.
	NAN *<-lang=c99>*	Defined when **float**-type **qNaN** is supported.
	FP_INFINITE *<-lang=c99>* FP_NAN *<-lang=c99>* FP_NORMAL *<-lang=c99>* FP_SUBNORMAL *<-lang=c99>* FP_ZERO *<-lang=c99>*	These indicate exclusive types of floating-point values.
	FP_ILOGB0 *<-lang=c99>* FP_ILOGBNAN *<-lang=c99>*	These are expanded to an integer constant expression of the value returned by **ilogb** when they are 0 or not-a-number, respectively.
	MATH_ERRNO *<-lang=c99>* MATH_ERREXCEPT *<-lang=c99>*	These are expanded to integer constants 1 and 2, respectively.
	math_errhandling *<-lang=c99>*	Expanded to an **int**-type expression whose value is a bitwise logical OR of **MATH_ERRNO** and **MATH_ERREXCEPT**.
Type	float_t *<-lang=c99>* double_t *<-lang=c99>*	These are floating-point types having the same width as **float** and **double**, respectively.

Type	Definition Name	Description
Function (macro)	fpclassify *<-lang=c99>*	Classifies argument values into not-a-number, infinity, normalized number, denormalized number, and 0.
	isfinite *<-lang=c99>*	Determines whether the argument is a finite value.
	isinf *<-lang=c99>*	Determines whether the argument is infinity.
	isnan *<-lang=c99>*	Determines whether the argument is a not-a-number.
	isnormal *<-lang=c99>*	Determines whether the argument is a normalized number.
	signbit *<-lang=c99>*	Determines whether the sign of the argument is negative.
	isgreater *<-lang=c99>*	Determines whether the first argument is greater than the second argument.
	isgreaterequal *<-lang=c99>*	Determines whether the first argument is equal to or greater than the second argument.
	isless *<-lang=c99>*	Determines whether the first argument is smaller than the second argument.
	islessequal *<-lang=c99>*	Determines whether the first argument is equal to or smaller than the second argument.
	islessgreater *<-lang=c99>*	Determines whether the first argument is smaller or greater than the second argument.
	isunordered *<-lang=c99>*	Determines whether the arguments are not ordered.
Function	acos / acosf / acosl	Calculates the arc cosine of a floating-point number.
	asin / asinf / asinl	Calculates the arc sine of a floating-point number.
	atan / atanf / atanl	Calculates the arc tangent of a floating-point number.
	atan2 / atan2f / atan2l	Calculates the arc tangent of the result of a division of two floating-point numbers.
	cos / cosf / cosl	Calculates the cosine of a floating-point radian value.
	sin / sinf / sinl	Calculates the sine of a floating-point radian value.
	tan / tanf / tanl	Calculates the tangent of a floating-point radian value.
	cosh / coshf / coshl	Calculates the hyperbolic cosine of a floating-point number.
	sinh / sinhf / sinhl	Calculates the hyperbolic sine of a floating-point number.
	tanh / tanhf / tanhl	Calculates the hyperbolic tangent of a floating-point number.
	exp / expf / expl	Calculates the exponential function of a floating-point number.
	frexp / frexpf / frexpl	Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.
	ldexp / ldexpf / ldexpl	Multiplies a floating-point number by a power of 2.
	log / logf / logl	Calculates the natural logarithm of a floating-point number.
	log10 / log10f / log10l	Calculates the base-ten logarithm of a floating-point number.
	modf / modff / modfl	Breaks a floating-point number into integral and fractional parts.
	pow / powf / powl	Calculates a power of a floating-point number.
	sqrt / sqrtf / sqrtl	Calculates the positive square root of a floating-point number.

Type	Definition Name	Description
Function	ceil / ceilf / ceill	Calculates the smallest integral value not less than or equal to the given floating-point number.
	fabs / fabsf / fabsl	Calculates the absolute value of a floating-point number.
	floor / floorf / floorl	Calculates the largest integral value not greater than or equal to the given floating-point number.
	fmod / fmodf / fmodl	Calculates the remainder of a division of two floating-point numbers.
	acosh / acoshf / acoshl *<-lang=c99>*	Calculates the hyperbolic arc cosine of a floating-point number.
	asinh / asinhf / asinhl *<-lang=c99>*	Calculates the hyperbolic arc sine of a floating-point number.
	atanh / atanhf / atanhl *<-lang=c99>*	Calculates the hyperbolic arc tangent of a floating-point number.
	exp2 / exp2f / exp2l *<-lang=c99>*	Calculates the value of 2 raised to the power **x**.
	expm1 / expm1f / expm1l *<-lang=c99>*	Calculates the natural logarithm raised to the power **x** and subtracts 1 from the result.
	ilogb / ilogbf / ilogbl *<-lang=c99>*	Extracts the exponent of **x** as a signed **int** value.
	log1p / log1pf / log1pl *<-lang=c99>*	Calculates the natural logarithm of the argument + 1.
	log2 / log2f / log2l *<-lang=c99>*	Calculates the base-2 logarithm.
	logb / logbf / logbl *<-lang=c99>*	Extracts the exponent of **x** as a signed integer.
	scalbn / scalbnf / scalbnl / scalbln / scalblnf / scalblnl *<-lang=c99>*	Calculates **x** × **FLT_RADIXn**.
	cbrt / cbrtf / cbrtl *<-lang=c99>*	Calculates the cube root of a floating-point number.
	hypot / hypotf / hypotl *<-lang=c99>*	Calculates the square root of the sum of squares of two parameters ($\sqrt{x^2 + y^2}$).
	erf / erff / erfl *<-lang=c99>*	Calculates the error function.
	erfc / erfcf / erfcl *<-lang=c99>*	Calculates the complementary error function.
	lgamma / lgammaf / lgammal *<-lang=c99>*	Calculates the natural logarithm of the absolute value of the gamma function.
	tgamma / tgammaf / tgammal *<-lang=c99>*	Calculates the gamma function.
	nearbyint / nearbyintf / nearbyintl *<-lang=c99>*	Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.
	rint / rintf / rintl *<-lang=c99>*	Equivalent to **nearbyint** except that this function group may generate floating-point exception.

Type	Definition Name	Description
Function	lrint / lrintf / lrintl / llrint / llrintf / llrintl <-lang=c99>	Rounds a floating-point number to the nearest integer according to the rounding direction.
	round / roundf / roundl <-lang=c99>	Rounds a floating-point number to the nearest integer in the floating-point representation.
	lround / lroundf / lroundl / llround / llroundf / llroundl <-lang=c99>	Rounds a floating-point number to the nearest integer.
	trunc / truncf / truncl <-lang=c99>	Rounds a floating-point number to the nearest integer in the floating-point representation.
	remainder / remainderf / remainderl <-lang=c99>	Calculates remainder **x REM y** specified in the IEEE60559 standard.
	remquo / remquof / remquol <-lang=c99>	Calculates the value having the same sign as **x/y** and the absolute value congruent modulo-2^n to the absolute value of the quotient.
	copysign / copysignf / copysignl <-lang=c99>	Generates a value consisting of the given absolute value and sign.
	nan / nanf / nanl <-lang=c99>	**nan("n string")** is equivalent to **("NAN(n string)"**, **(char**) NULL)**.
	nextafter / nextafterf / nextafterl <-lang=c99>	Converts a floating-point number to the type of the function and calculates the representable value following the converted number on the real axis.
	nexttoward / nexttowardf / nexttowardl <-lang=c99>	Equivalent to the **nextafter** function group except that the second argument is of type long double and returns the second argument after conversion to the type of the function.
	fdim / fdimf / fdiml <-lang=c99>	Calculates the positive difference.
	fmax / fmaxf / fmaxl <-lang=c99>	Obtains the greater of two values.
	fmin / fminf / fminl <-lang=c99>	Obtains the smaller of two values.
	fma / fmaf / fmal <-lang=c99>	Calculates **(d1 * d2) + d3** as a single ternary operation.

Operation in the event of an error is described below.

(1) Domain error
A domain error occurs if the value of a parameter input to a function is outside the domain over which the mathematical function is defined. In this case, the value of **EDOM** is set in **errno**. The function return value in implementation-defined.

(2) Range error
A range error occurs if the result of a function cannot be represented as a value of the double type. In this case, the value of **ERANGE** is set in **errno**. If the result overflows, the function returns the value of **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** with the same sign as the correct value of the function. If the result underflows, 0 is returned as the return value.

Notes 1. If there is a possibility of a domain error resulting from a **<math.h>** function call, it is dangerous to use the resultant value directly. The value of **errno** should always be checked before using the result in such cases.

[Format]

```
      .
      .
      .
1   x=asin(a);
2   if (errno==EDOM)
3       printf ("error\n");
4   else
5       printf ("result is : %lf\n",x);
      .
      .
      .
```

In line 1, the arc sine value is computed using the **asin** function. If the value of argument **a** is outside the **asin** function domain [–1.0, 1.0], the **EDOM** value is set in **errno**. Line 2 determines whether a domain error has occurred. If a domain error has occurred, **error** is output in line 3. If there is no domain error, the arc sine value is output in line 5.

Notes 2. Whether or not a range error occurs depends on the internal representation format of floating-point types determined by the compiler. For example, if an internal representation format that allows an infinity to be represented as a value is used, **<math.h>** library functions can be implemented without causing range errors.

Implementation-Defined Specifications

Item	Compiler Specifications
Value returned by a mathematical function if an input argument is out of the range	A not-a-number is returned. For details on the format of not-a-numbers, refer to section "(5) Floating-Point Number Specifications" in "4.1.5 Internal Data Representation and Areas".
Whether **errno** is set to the value of macro **ERANGE** if an underflow error occurs in a mathematical function	For the functions that set **errno** to the value of **ERANGE**, see "10.5.6 Standard Library Error Messages". The other functions do not set **errno** to **ERANGE**.
Whether a range error occurs if the second argument in the **fmod** function is 0	A range error occurs. For details of the return value from **fmod**, see "fmod / fmodf / fmodl" in "7.4.7 <math.h>".

acos / acosf / acosl

Calculates the arc cosine of a floating-point number.

[Format]

```
#include <math.h>
double acos (double d);
float acosf (float d);
long double acosl (long double d);
```

[Parameters]

d Floating-point number for which arc cosine is to be computed

[Return values]

Normal: Arc cosine of d
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs for a value of d not in the range [–1.0, +1.0].
The acos function returns the arc cosine in the range [0, π] by the radian.
If the function call is replaced by code that uses a trigonometric function unit, the function might no longer be reentrant. For details, refer to the item about **-tfu** in the section on compile options.

asin / asinf / asinl

Calculates the arc sine of a floating-point number.

[Format]

```
#include <math.h>
double asin (double d);
float asinf (float d);
long double asinl (long double);
```

[Parameters]

d Floating-point number for which arc sine is to be computed

[Return values]

Normal: Arc sine of d
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs for a value of d not in the range [–1.0, +1.0].
The asin function returns the arc sine in the range [–π/2, +π/2] by the radian.
If the function call is replaced by code that uses a trigonometric function unit, the function might no longer be reentrant. For details, refer to the item about **-tfu** in the section on compile options.

atan / atanf / atanl

Calculates the arc tangent of a floating-point number.

[Format]

```
#include <math.h>
double atan (double d);
float atanf (float d);
long double atanl (long double d);
```

[Parameters]

d Floating-point number for which arc tangent is to be computed

[Return values]

Arc tangent of d

[Remarks]

The atan function returns the arc tangent in the range ($-\pi/2$, $+\pi/2$) by the radian.

If the function call is replaced by code that uses a trigonometric function unit, the function might no longer be reentrant. For details, refer to the item about **-tfu** in the section on compile options.

atan2 / atan2f / atan2l

Calculates the arc tangent of the division of two floating-point numbers.

[Format]

```
#include <math.h>
double atan2 (double y, double x);
float atan2f (float y, float x);
long double atan2l (long double y, long double x);
```

[Parameters]

x Divisor
y Dividend

[Return values]

Normal: Arc tangent value when y is divided by x
Abnormal: Domain error: Returns not-a-number.

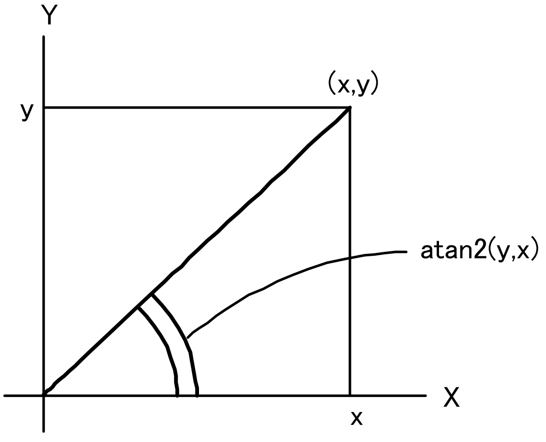
[Remarks]

A domain error occurs if the values of both x and y are 0.0.

The atan2 function returns the arc tangent in the range ($-\pi$, $+\pi$) by the radian. The meaning of the atan2 function is illustrated in Figure 7.1. As shown in the figure, the result of the atan2 function is the angle between the X-axis and a straight line passing through the origin and point (x, y).

If y = 0.0 and x is negative, the result is π. If x = 0.0, the result is $\pm\pi/2$, depending on whether y is positive or negative. However, a divide-by-zero exception might occur depending on the microcomputer settings.

Figure 7.1 Meaning of atan2 Function



cos / cosf / cosl

Calculates the cosine of a floating-point radian value.

[Format]

```
#include <math.h>
double cos (double d);
float cosf (float d);
long double cosl (long double d);
```

[Parameters]

d Radian value for which cosine is to be computed

[Return values]

Cosine of d

sin / sinf / sinl

Calculates the sine of a floating-point radian value.

[Format]

```
#include <math.h>
double sin (double d);
float sinf (float d);
long double sinl (long double d);
```

[Parameters]

d Radian value for which sine is to be computed

[Return values]

Sine of d

tan / tanf / tanl

Calculates the tangent of a floating-point radian value.

[Format]

```
#include <math.h>
double tan (double d);
float tanf (float d);
long double tanl (long double d);
```

[Parameters]

d Radian value for which tangent is to be computed

[Return values]

Tangent of d

[Remarks]

If the function call is replaced by code that uses a trigonometric function unit, the function might no longer be reentrant. For details, refer to the item about **-tfu** in the section on compile options.

cosh / coshf / coshl

Calculates the hyperbolic cosine of a floating-point number.

[Format]

```
#include <math.h>
double cosh (double d);
float coshf (float d);
long double coshl (long double d);
```

[Parameters]

d Floating-point number for which hyperbolic cosine is to be computed

[Return values]

Hyperbolic cosine of d

sinh / sinhf / sinhl

Calculates the hyperbolic sine of a floating-point number.

[Format]

```
#include <math.h>
double sinh (double d);
float sinhf (float d);
long double sinhl (long double d);
```

[Parameters]

d Floating-point number for which hyperbolic sine is to be computed

[Return values]

Hyperbolic sine of d

tanh / tanhf / tanhl

Calculates the hyperbolic tangent of a floating-point number.

[Format]

```
#include <math.h>
double tanh (double d);
float tanhf (float d);
long double tanhl (long double d);
```

[Parameters]

d Floating-point number for which hyperbolic tangent is to be computed

[Return values]

Hyperbolic tangent of d

exp / expf / expl

Calculates the exponential function of a floating-point number.

[Format]

```
#include <math.h>
double exp (double d);
float expf (float d);
long double expl (long double d);
```

[Parameters]

d Floating-point number for which exponential function is to be computed

[Return values]

Exponential function value of d

frexp / frexpf / frexpl

Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.

[Format]

```
#include <math.h>
double frexp (double value, long *exp);
float frexpf (float value, long * exp);
long double frexpl (long double value, long *exp);
```

[Parameters]

value Floating-point number to be broken into a [0.5, 1.0) value and a power of 2
exp Pointer to storage area that holds power-of-2 value

[Return values]

If value is 0.0: 0.0
If value is not 0.0: Value of ret defined by ret * $2^{\text{value pointed to by exp}}$ = value

[Remarks]

The frexp function breaks value into a [0.5, 1.0) value and a power of 2. It stores the resultant power-of-2 value in the area pointed to by exp.

The frexp function returns the return value ret in the range [0.5, 1.0) or as 0.0.

If value is 0.0, the contents of the int storage area pointed to by exp and the value of ret are both 0.0.

ldexp / ldexpf / ldexpl

Multiplies a floating-point number by a power of 2.

[Format]

```
#include <math.h>
double ldexp (double e, long f);
float ldexpf (float e, long f);
long double ldexpl (long double e, long f);
```

[Parameters]

e Floating-point number to be multiplied by a power of 2
f Power-of-2 value

[Return values]

Result of $e * 2^f$ operation

log / logf / logl

Calculates the natural logarithm of a floating-point number.

[Format]

```
#include <math.h>
double log (double d);
float logf (float d);
long double logl (long double d);
```

[Parameters]

d Floating-point number for which natural logarithm is to be computed

[Return values]

Normal: Natural logarithm of d
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if d is negative.
A range error occurs if d is 0.0.

log10 / log10f / log10l

Calculates the base-ten logarithm of a floating-point number.

[Format]

```
#include <math.h>
double log10 (double d);
float log10f(float d);
long double log10l(long double d);
```

[Parameters]

d Floating-point number for which base-ten logarithm is to be computed

[Return values]

Normal: Base-ten logarithm of d
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if d is negative.
A range error occurs if d is 0.0.

modf / modff / modfl

Breaks a floating-point number into integral and fractional parts.

[Format]

```
#include <math.h>
double modf (double a, double*b);
float modff (float a, float *b);
long double modfl (long double a, long double *b);
```

[Parameters]

a Floating-point number to be broken into integral and fractional parts
b Pointer indicating storage area that stores integral part

[Return values]

Fractional part of a

pow / powf / powl

Calculates a power of floating-point number.

[Format]

```
#include <math.h>
double pow (double x, double y);
float powf (float x, float y);
long double powl (long double x, long double y);
```

[Parameters]

x Value to be raised to a power
y Power value

[Return values]

Normal: Value of x raised to the power y
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if x is 0.0 and y is 0.0 or less, or if x is negative and y is not an integer.

sqrt / sqrtf / sqrtl

Calculates the positive square root of a floating-point number.

[Format]

```
#include <math.h>
double sqrt (double d);
float sqrtf (float d);
long double sqrtl (long double d);
```

[Parameters]

d Floating-point number for which positive square root is to be computed

[Return values]

Normal: Positive square root of d
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if d is negative.

ceil / ceilf / ceill

Returns the smallest integral value not less than or equal to the given floating-point number.

[Format]

```
#include <math.h>
double ceil (double d);
float ceilf (float d);
long double ceill (long double d);
```

[Parameters]

d Floating-point number for which smallest integral value not less than that number is to be computed

[Return values]

Smallest integral value not less than or equal to d

[Remarks]

The ceil function returns the smallest integral value not less than or equal to d, expressed as a double type value. Therefore, if d is negative, the value after truncation of the fractional part is returned.

fabs / fabsf / fabsl

Calculates the absolute value of a floating-point number.

[Format]

```
#include <math.h>
double fabs (double d);
float fabsf (float d);
long double fabsl (long double d);
```

[Parameters]

d Floating-point number for which absolute value is to be computed

[Return values]

Absolute value of d

floor / floorf / floorl

Returns the largest integral value not greater than or equal to the given floating-point number.

[Format]

```
#include <math.h>
double floor (double d);
float floorf (float d);
long double floorl (long double d);
```

[Parameters]

d Floating-point number for which largest integral value not greater than that number is to be computed

[Return values]

Largest integral value not greater than or equal to d

[Remarks]

The floor function returns the largest integral value not greater than or equal to d, expressed as a double type value. Therefore, if d is negative, the value after rounding-up of the fractional part is returned.

fmod / fmodf / fmodl

Calculates the remainder of a division of two floating-point numbers.

[Format]

```
#include <math.h>
double fmod (double x, double y);
float fmodf (float x, float y);
long double fmodl (long double x, long double y);
```

[Parameters]

x Dividend
y Divisor

[Return values]

When y is 0.0: x
When y is not 0.0: Remainder of division of x by y
When y is ±∞: Returns x.
When x is ±∞ or y is 0: Returns not-a-number and sets global variable **errno** to the value of macro **EDOM**.

[Remarks]

In the fmod function, the relationship between parameters x and y and return value ret is as follows:
x = y * i + ret (where i is an integer)
The sign of return value ret is the same as the sign of x.
If the quotient of x/y cannot be represented, the value of the result is not guaranteed.

acosh / acoshf / acoshl

Calculates the hyperbolic arc cosine of a floating-point number.

[Format]

```
#include <math.h>
double acosh(double d);
float acoshf(float d);
long double acoshl(long double d);
```

[Parameters]

d Floating-point number for which hyperbolic arc cosine is to be computed

[Return values]

Normal: Hyperbolic arc cosine of d
Abnormal: Domain error: Returns NaN.
Error conditions: A domain error occurs when d is smaller than 1.0.

[Remarks]

The acosh function returns the hyperbolic arc cosine in the range [0, +∞].

asinh / asinhf / asinhl

Calculates the hyperbolic arc sine of a floating-point number.

[Format]

```
#include <math.h>
double asinh(double d);
float asinhf(float d);
long double asinhl(long double d);
```

[Parameters]

d Floating-point number for which hyperbolic arc sine is to be computed

[Return values]

Hyperbolic arc sine of d

atanh / atanhf / atanhl

Calculates the hyperbolic arc tangent of a floating-point number.

[Format]

```
#include <math.h>
double atanh(double d);
float atanhf(float d);
long double atanhl(long double d);
```

[Parameters]

d Floating-point number for which hyperbolic arc tangent is to be computed

[Return values]

Normal: Hyperbolic arc tangent of d
Abnormal: Domain error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL depending on the function.
Range error: Returns not-a-number.

[Remarks]

A domain error occurs for a value of d not in the range [–1, +1]. A range error may occur for a value of d equal to –1 or 1.

exp2 / exp2f / exp2l

Calculates the value of 2 raised to the power d.

[Format]

```
#include <math.h>
double exp2(double d);
float exp2f(float d);
long double exp2l(long double d);
```

[Parameters]

d Floating-point number for which exponential function is to be computed

[Return values]

Normal: Exponential function value of 2
Abnormal: Range error: Returns 0, or returns +HUGE_VAL, +HUGE_VALF, or +HUGE_VALL depending on the function

[Remarks]

A range error occurs if the absolute value of d is too large.

expm1 / expm1f / expm1l

Calculates the value of natural logarithm base e raised to the power d and subtracts 1 from the result.

[Format]

```
#include <math.h>
double expm1(double d);
float expm1f(float d);
long double expm1l(long double d);
```

[Parameters]

d Power value to which natural logarithm base e is to be raised

[Return values]

Normal: Value obtained by subtracting 1 from natural logarithm base e raised to the power d
Abnormal: Range error: Returns -HUGE_VAL, –HUGE_VALF, or –HUGE_VALL depending on the function.

[Remarks]

expm1(d) provides more accurate calculation than exp(x) – 1 even when d is near to 0.

ilogb / ilogbf / ilogbl

Extracts the exponent of d.

[Format]

```
#include <math.h>
long ilogb(double d);
long ilogbf(float d);
long ilogbl(long double d);
```

[Parameters]

d Value of which exponent is to be extracted

[Return values]

Normal: Exponential function value of d
d is ∞: INT_MAX
d is not-a-number: FP_ILOGBNAN
d is 0: FP_ILOGBNAN
Abnormal: d is 0 and a range error has occurred: FP_ILOGB0

[Remarks]

A range error may occur if d is 0.

log1p / log1pf / log1pl

Calculates the natural logarithm (base e) of d + 1.

[Format]

```
#include <math.h>
double log1p(double d);
float log1pf(float d);
long double log1pl(long double d);
```

[Parameters]

d Value for which the natural logarithm of this parameter + 1 is to be computed

[Return values]

Normal: Natural logarithm of d + 1
Abnormal: Domain error: Returns not-a-number.
Range error: Returns –HUGE_VAL, –HUGE_VALF, or –HUGE_VALL depending on the function.

[Remarks]

A domain error occurs if d is smaller than –1.
A range error occurs if d is –1.
log1p(d) provides more accurate calculation than log(1+d) even when d is near to 0.

log2 / log2f / log2l

Calculates the base-2 logarithm of d.

[Format]

```
#include <math.h>
double log2(double d);
float log2f(float d);
long double log2l(long double d);
```

[Parameters]

d Value of which logarithm is to be calculated

[Return values]

Normal: Base-2 logarithm of d
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if d is a negative value.

logb / logbf / logbl

Extracts the exponent of d in internal floating-point representation, as a floating-point value.

[Format]

```
#include <math.h>
double logb(double d);
float logbf(float d);
long double logbl(long double d);
```

[Parameters]

d Value of which exponent is to be extracted

[Return values]

Normal: Signed exponent of d
Abnormal: Range error: Returns –HUGE_VAL, –HUGE_VALF, or –HUGE_VALL depending on the function.

[Remarks]

A range error may occur if d is 0.
d is always assumed to be normalized.

scalbn / scalbnf / scalbnl / scalbln / scalblnf / scalblnl

Calculates a floating-point number multiplied by a power of radix, which is an integer.

[Format]

```
#include <math.h>
double scalbn(double d, long e);
float scalbnf(float d, long e);
long double scalbnl(long double d, long e);
double scalbln(double d, long e);
float scalblnf(float d, long int e);
long double scalblnl(long double d, long int e);
```

[Parameters]

d Value to be multiplied by FLT_RADIX raised to the power e
e Exponent used to compute a power of FLT_RADIX

[Return values]

Normal: Value equal to d multiplied by FLT_RADIX
Abnormal: Range error: Returns –HUGE_VAL, –HUGE_VALF, or –HUGE_VALL depending on the function.

[Remarks]

A range error may occur if d is 0.
FLT_RADIX raised to the power e is not actually calculated.

cbrt / cbrtf / cbrtl

Calculates the cube root of a floating-point number.

[Format]

```
#include <math.h>
double cbrt(double d);
float cbrtf(float d);
long double cbrtl(long double d);
```

[Parameters]

d Value for which a cube root is to be computed

[Return values]

Cube root of d

hypot / hypotf / hypotl

Calculates the square root of the sum of squares of floating-point values ($\sqrt{x^2 + y^2}$).

[Format]

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

[Parameters]

x, y Values for which the square root of the sum of squares of two parameters ($\sqrt{x^2 + y^2}$) is to be calculated

[Return values]

Normal: Square root of the sum of squares of two parameters ($\sqrt{x^2 + y^2}$)
Abnormal: Range error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL depending on the function.

[Remarks]

A range error may occur if the result overflows.

erf / erff / erfl

Calculates the error function value of a floating-point number.

[Format]

```
#include <math.h>
double erf(double d);
float erff(float d);
long double erfl(long double d);
```

[Parameters]

d Value for which the error function value is to be computed

[Return values]

Error function value of d

erfc / erfcf / erfcl

Calculates the complementary error function value of a floating-point number.

[Format]

```
#include <math.h>
double erfc(double d);
float erfcf(float d);
long double erfcl(long double d);
```

[Parameters]

d Value for which the complementary error function value is to be computed

[Return values]

Complementary error function value of d

[Remarks]

A range error occurs if the absolute value of d is too large.

lgamma / lgammaf / lgammal

Calculates the logarithm of the gamma function of a floating-point number.

[Format]

```
#include <math.h>
double lgamma(double d);
float lgammaf(float d);
long double lgammal(long double d);
```

[Parameters]

d Value for which the logarithm of the gamma function is to be computed

[Return values]

Normal: Logarithm of gamma function of d
Abnormal: Domain error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL with the mathematically correct sign.
Range error: Returns +HUGE_VAL, +HUGE_VALF, or +HUGE_VALL.

[Remarks]

A range error is set if the absolute value of d is too large or small.
A domain error occurs if d is a negative integer or 0 and the calculation result is not representable.

tgamma / tgammaf / tgammal

Calculates the gamma function of a floating-point number.

[Format]

```
#include <math.h>
double tgamma(double d);
float tgammaf(float d);
long double tgammal(long double d);
```

[Parameters]

d Value for which the gamma function value is to be computed

[Return values]

Normal: Gamma function value of d
Abnormal: Domain error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL with the same sign as that of d.
Range error: Returns 0, or returns +HUGE_VAL, +HUGE_VALF, or +HUGE_VALL with the mathematically correct sign depending on the function.

[Remarks]

A range error is set if the absolute value of d is too large or small.
A domain error occurs if d is a negative integer or 0 and the calculation result is not representable.

nearbyint / nearbyintf / nearbyintl

Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.

[Format]

```
#include <math.h>
double nearbyint(double d);
float nearbyintf(float d);
long double nearbyintl(long double d);
```

[Parameters]

d Value to be rounded to an integer in the floating-point format

[Return values]

d rounded to an integer in the floating-point format

[Remarks]

The nearbyint function group does not generate "inexact" floating-point exceptions.

rint / rintf / rintl

Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.

[Format]

```
#include <math.h>
double rint(double d);
float rintf(float d);
long double rintl(long double d);
```

[Parameters]

d Value to be rounded to an integer in the floating-point format

[Return values]

d rounded to an integer in the floating-point format

[Remarks]

The rint function group differs from the nearbyint function group only in that the ring function group may generate "inexact" floating-point exceptions.

lrint / lrintf / lrintl / llrint / llrintf / llrintl

Rounds a floating-point number to the nearest integer according to the current rounding direction.

[Format]

```
#include <math.h>
long int lrint(double d);
long int lrintf(float d);
long int lrintl(long double d);
long long int llrint(double d);
long long int llrintf(float d);
long long int llrintl(long double d);
```

[Parameters]

d Value to be rounded to an integer

[Return values]

Normal: d rounded to an integer
Abnormal: Range error: Returns an undetermined value.

[Remarks]

A range error may occur if the absolute value of d is too large.
The return value is unspecified when the rounded value is not in the range of the return value type.

round / roundf / roundl

Rounds a floating-point number to the nearest integer in the floating-point representation.

[Format]

```
#include <math.h>
double round(double d);
float roundf(float d);
long double roundl(long double d);
```

[Parameters]

d Value to be rounded to an integer

[Return values]

Normal: d rounded to an integer
Abnormal: Range error: Returns an undetermined value.

[Remarks]

A range error may occur if the absolute value of d is too large.

lround / lroundf / lroundl / llround / llroundf / llroundl

Rounds a floating-point number to the nearest integer.

[Format]

```
#include <math.h>
long int lround(double d);
long int lroundf(float d);
long int lroundl(long double d);
long long int llround (double d);
long long int llroundf(float d);
long long int llroundl(long double d);
```

[Parameters]

d Value to be rounded to an integer

[Return values]

Normal: d rounded to an integer
Abnormal: Range error: Returns an undetermined value.

[Remarks]

A range error may occur if the absolute value of d is too large.

When d is at the midpoint between two integers, the lround function group selects the integer farther from 0 regardless of the current rounding direction. The return value is unspecified when the rounded value is not in the range of the return value type.

trunc / truncf / truncl

Rounds a floating-point number to the nearest integer in the floating-point representation.

[Format]

```
#include <math.h>
double trunc(double d);
float truncf(float d);
long double truncl(long double d);
```

[Parameters]

d Value to be rounded to an integer in the floating-point representation

[Return values]

d truncated to an integer in the floating-point format

[Remarks]

The trunc function group rounds d so that the absolute value after rounding is not greater than the absolute value of d.

remainder / remainderf / remainderl

Calculates the remainder of a division of two floating-point numbers.

[Format]

```
#include <math.h>
double remainder(double d1, double d2);
float remainderf(float d1, float d2);
long double remainderl(long double d1, long double d2);
```

[Parameters]

d1, d2 Values for which remainder of a division is to be computed (d1 / d2)

[Return values]

Remainder of division of d1 by d2

[Remarks]

The remainder calculation by the remainder function group conforms to the IEEE 60559 standard.

remquo / remquof / remquol

Calculates the remainder of a division of two floating-point numbers.

[Format]

```
#include <math.h>
double remquo(double d1, double d2, long *q);
float remquof(float d1, float d2, long *q);
long double remquol(long double d1, long double d2, long *q);
```

[Parameters]

d1, d2 Values for which remainder of a division is to be computed (d1 / d2)
q Value pointing to the location to store the quotient obtained by remainder calculation

[Return values]

Remainder of division of d1 by d2

[Remarks]

The value stored in the location indicated by q has the same sign as the result of x/y and the integral quotient of modulo-2n x/y (n is an implementation-defined integer equal to or greater than 3).

copysign / copysignf / copysignl

Generates a value consisting of the absolute value of d1 and the sign of d2.

[Format]

```
#include <math.h>
double copysign(double d1, double d2);
float copysignf(float d1, float d2);
long double copysignl(long double d1, long double d2);
```

[Parameters]

d1 Value of which absolute value is to be used in the generated value
d2 Value of which sign is to be used in the generated value

[Return values]

Normal: Value consisting of absolute value of d1 and sign of d2
Abnormal: Range error: Returns an undetermined value.

[Remarks]

When d1 is a not-a-number, the copysign function group generates a not-a-number with the sign bit of d2.

nan / nanf / nanl

Returns not-a-number.

[Format]

```
#include <math.h>
double nan(const char *c);
float nanf(const char *c);
long double nanl(const char *c);
```

[Parameters]

c Pointer to a string

[Return values]

qNaN with the contents of the location indicated by c or 0 (when qNaN is not supported)

[Remarks]

The nan("c string") call is equivalent to strtod("NAN(c string)", (char**) NULL). The nanf and nanl calls are equivalent to the corresponding strtof and strtold calls, respectively.

nextafter / nextafterf / nextafterl

Calculates the next floating-point representation following d1 in the direction to d2 on the real axis.

[Format]

```
#include <math.h>
double nextafter(double d1, double d2);
float nextafterf(float d1, float d2);
long double nextafterl(long double d1, long double d2);
```

[Parameters]

d1 Floating-point value on the real axis
d2 Value indicating the direction viewed from d1, in which a representable floating-point value is to be found

[Return values]

Normal: Representable floating-point value
Abnormal: Range error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL with the mathematically correct sign depending on the function.

[Remarks]

A range error may occur if d1 is the maximum finite value that can be represented in its type and the return value is an infinity or cannot be represented in its type.
The nextafter function group returns d2 when d1 is equal to d2.

nexttoward / nexttowardf / nexttowardl

Calculates the next floating-point representation following d1 in the direction to d2 on the real axis.

[Format]

```
#include <math.h>
double nexttoward(double d1, long double d2);
float nexttowardf(float d1, long double d2);
long double nexttowardl(long double d1, long double d2);
```

[Parameters]

d1 Floating-point value on the real axis
d2 Value indicating the direction viewed from d1, in which a representable floating-point value is to be found

[Return values]

Normal: Representable floating-point value
Abnormal: Range error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL with the mathematically correct sign depending on the function

[Remarks]

A range error may occur if d1 is the maximum finite value that can be represented in its value and the return value is an infinity or cannot be represented in its type.
The nexttoward function group is equivalent to the nextafter function group except that d2 is of type long double and returns d2 after conversion depending of the function when d1 is equal to d2.

fdim / fdimf / fdiml

Calculates the positive difference between two arguments.

[Format]

```
#include <math.h>
double fdim(double d1, double d2);
float fdimf(float d1, float d2);
long double fdiml(long double d1, long double d2);
```

[Parameters]

d1, d2 Values of which difference is to be computed (|d1 - d2|)

[Return values]

Normal: Positive difference between two arguments
Abnormal: Range error: HUGE_VAL, HUGE_VALF, or HUGE_VALL

[Remarks]

A range error may occur if the return value overflows.

fmax / fmaxf / fmaxl

Obtains the greater of two arguments.

[Format]

```
#include <math.h>
double fmax(double d1, double d2);
float fmaxf(float d1, float d2);
long double fmaxl(long double d1, long double d2);
```

[Parameters]

d1, d2 Values to be compared

[Return values]

Greater of two arguments

[Remarks]

The fmax function group recognizes a not-a-number as a lack of data. When one argument is a not-a-number and the other is a numeric value, the function returns the numeric value.

fmin / fminf / fminl

Obtains the smaller of two arguments.

[Format]

```
#include <math.h>
double fmin(double d1, double d2);
float fminf(float d1, float d2);
long double fminl(long double d1, long double d2);
```

[Parameters]

d1, d2 Values to be compared

[Return values]

Smaller of two arguments

[Remarks]

The fmin function group recognizes a not-a-number as a lack of data. When one argument is a not-a-number and the other is a numeric value, the function returns the numeric value.

fma / fmaf / fmal

Calculates (d1 * d2) + d3 as a single ternary operation.

[Format]

```
#include <math.h>
double fma(double d1, double d2, double d3);
float fmaf(float d1, float d2, float d3);
long double fmal(long double d1, long double d2, long double d3);
```

[Parameters]

d1, d2, d3 Floating-point values

[Return values]

Result of (d1 * d2) + d3 calculated as ternary operation

[Remarks]

The fma function group performs calculation as if infinite precision is available and rounds the result only one time in the rounding mode indicated by FLT_ROUNDS.

7.4.8 <mathf.h>

Performs various mathematical operations.

<mathf.h> declares mathematical functions and defines macros in single-precision format. The mathematical functions and macros used here do not follow the ANSI specifications. Each function receives **float**-type arguments and returns a **float**-type value.

The following constants (macros) are all implementation-defined.

Type	Definition Name	Description
Constant (macro)	EDOM	Indicates the value to be set in **errno** if the value of a parameter input to a function is outside the range of values defined in the function.
	ERANGE	Indicates the value to be set in **errno** if the result of a function cannot be represented as a **float** type value, or if an overflow or an underflow occurs.
	HUGE_VALF HUGE_VAL HUGE_VALL	Indicates the value for the function return value if the result of a function overflows.
Function	acosf	Calculates the arc cosine of a floating-point number.
	asinf	Calculates the arc sine of a floating-point number.
	atanf	Calculates the arc tangent of a floating-point number.
	atan2f	Calculates the arc tangent of the result of a division of two floating-point numbers.
	cosf	Calculates the cosine of a floating-point radian value.
	sinf	Calculates the sine of a floating-point radian value.
	tanf	Calculates the tangent of a floating-point radian value.
	coshf	Calculates the hyperbolic cosine of a floating-point number.
	sinhf	Calculates the hyperbolic sine of a floating-point number.
	tanhf	Calculates the hyperbolic tangent of a floating-point number.
	expf	Calculates the exponential function of a floating-point number.
	frexpf	Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.
	ldexpf	Multiplies a floating-point number by a power of 2.
	logf	Calculates the natural logarithm of a floating-point number.
	log10f	Calculates the base-ten logarithm of a floating-point number.
	modff	Breaks a floating-point number into integral and fractional parts.
	powf	Calculates a power of a floating-point number.
	sqrtf	Calculates the positive square root of a floating-point number.
	ceilf	Calculates the smallest integral value not less than or equal to the given floating-point number.
	fabsf	Calculates the absolute value of a floating-point number.
	floorf	Calculates the largest integral value not greater than or equal to the given floating-point number.
	fmodf	Calculates the remainder of a division of two floating-point numbers.

Operation in the event of an error is described below.

(1) Domain error
A domain error occurs if the value of a parameter input to a function is outside the domain over which the mathematical function is defined. In this case, the value of **EDOM** is set in **errno**. The function return value in implementation-defined.

(2) Range error
A range error occurs if the result of a function cannot be represented as a **float** type value. In this case, the value of **ERANGE** is set in **errno**. If the result overflows, the function returns the value of **HUGE_VALF**, with the same sign as the correct value of the function. If the result underflows, 0 is returned as the return value.

Notes 1. If there is a possibility of a domain error resulting from a **<mathf.h>** function call, it is dangerous to use the resultant value directly. The value of **errno** should always be checked before using the result in such cases.

[Format]

```
      .
      .
      .
1  x=asinf(a);
2  if (errno==EDOM)
3      printf ("error\n");
4  else
5      printf ("result is : %f\n",x);
      .
      .
      .
```

In line 1, the arc sine value is computed using the **asinf** function. If the value of argument **a** is outside the **asinf** function domain [–1.0, 1.0], the **EDOM** value is set in **errno**. Line 2 determines whether a domain error has occurred. If a domain error has occurred, error is output in line 3. If there is no domain error, the arc sine value is output in line 5.

Notes 2. Whether or not a range error occurs depends on the internal representation format of floating-point types determined by the compiler. For example, if an internal representation format that allows an infinity to be represented as a value is used, **<mathf.h>** library functions can be implemented without causing range errors.

Implementation-Defined Specifications

Item	Compiler Specifications
Value returned by a mathematical function if an input argument is out of the range	A not-a-number is returned. For details on the format of not-a-numbers, refer to section 4.1.6 (5) Floating-Point Number Specifications.
Whether **errno** is set to the value of macro **ERANGE** if an underflow error occurs in a mathematical function	Not specified
Whether a range error occurs if the second argument in the **fmodf** function is 0	A range error occurs.

acosf

Calculates the arc cosine of a floating-point number.

[Format]

```
#include <mathf.h>
float acosf (float f);
```

[Parameters]

f Floating-point number for which arc cosine is to be computed

[Return values]

Normal: Arc cosine of f
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs for a value of f not in the range [–1.0, +1.0]
The acosf function returns the arc cosine in the range [0, π] by the radian.

asinf

Calculates the arc sine of a floating-point number.

[Format]

```
#include <mathf.h>
float asinf (float f);
```

[Parameters]

f Floating-point number for which arc sine is to be computed

[Return values]

Normal: Arc sine of f
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs for a value of f not in the range [–1.0, +1.0].
The asinf function returns the arc sine in the range [–π/2, +π/2] by the radian.

atanf

Calculates the arc tangent of a floating-point number.

[Format]

```
#include <mathf.h>
float atanf (float f);
```

[Parameters]

f Floating-point number for which arc tangent is to be computed

[Return values]

Arc tangent of f

[Remarks]

The atanf function returns the arc tangent in the range $(-\pi/2, +\pi/2)$ by the radian.

atan2f

Calculates the arc tangent of the division of two floating-point numbers.

[Format]

```
#include <mathf.h>
float atan2f (float y, float x);
```

[Parameters]

x Divisor
y Dividend

[Return values]

Normal: Arc tangent value when y is divided by x
Abnormal: Domain error: Returns not-a-number.
Error conditions: A domain error occurs if the values of both x and y are 0.0.

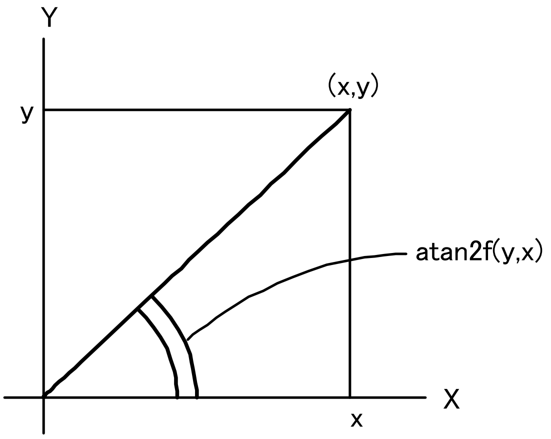
[Remarks]

A domain error occurs if the values of both x and y are 0.0.

The atan2f function returns the arc tangent in the range ($-\pi$, $+\pi$) by the radian. The meaning of the atan2f function is illustrated in Figure 7.2. As shown in the figure, the result of the atan2f function is the angle between the X-axis and a straight line passing through the origin and point (x, y).

If y = 0.0 and x is negative, the result is π. If x = 0.0, the result is $\pm\pi/2$, depending on whether y is positive or negative.

Figure 7.2 Meaning of atan2f Function



cosf

Calculates the cosine of a floating-point radian value.

[Format]

```
#include <mathf.h>
float cosf (float f);
```

[Parameters]

f Radian value for which cosine is to be computed

[Return values]

Cosine of f

sinf

Calculates the sine of a floating-point radian value.

[Format]

```
#include <mathf.h>
float sinf (float f);
```

[Parameters]

f Radian value for which sine is to be computed

[Return values]

Sine of f

tanf

Calculates the tangent of a floating-point radian value.

[Format]

```
#include <mathf.h>
float tanf (float f);
```

[Parameters]

f Radian value for which tangent is to be computed

[Return values]

Tangent of f

coshf

Calculates the hyperbolic cosine of a floating-point number.

[Format]

```
#include <mathf.h>
float coshf (float f);
```

[Parameters]

f Floating-point number for which hyperbolic cosine is to be computed

[Return values]

Hyperbolic cosine of f

sinhf

Calculates the hyperbolic sine of a floating-point number.

[Format]

```
#include <mathf.h>
float sinhf (float f);
```

[Parameters]

f Floating-point number for which hyperbolic sine is to be computed

[Return values]

Hyperbolic sine of f

tanhf

Calculates the hyperbolic tangent of a floating-point number.

[Format]

```
#include <mathf.h>
float tanhf (float f);
```

[Parameters]

f Floating-point number for which hyperbolic tangent is to be computed

[Return values]

Hyperbolic tangent of f

expf

Calculates the exponential function of a floating-point number.

[Format]

```
#include <mathf.h>
float expf (float f);
```

[Parameters]

f Floating-point number for which exponential function is to be computed

[Return values]

Exponential function value of f

frexpf

Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.

[Format]

```
#include <mathf.h>
float frexpf (float value, float long *exp);
```

[Parameters]

value Floating-point number to be broken into a [0.5, 1.0) value and a power of 2
exp Pointer to storage area that holds power-of-2 value

[Return values]

If value is 0.0: 0.0
If value is not 0.0: Value of ret defined by ret * $2^{\text{value pointed to by exp}}$ = value

[Remarks]

The frexpf function breaks value into a [0.5, 1.0) value and a power of 2. It stores the resultant power-of-2 value in the area pointed to by exp.

The frexpf function returns the return value ret in the range [0.5, 1.0) or as 0.0.

If value is 0.0, the contents of the int storage area pointed to by exp and the value of ret are both 0.0.

ldexpf

Multiplies a floating-point number by a power of 2.

[Format]

```
#include <mathf.h>
float ldexpf (float e, long f);
```

[Parameters]

e Floating-point number to be multiplied by a power of 2
f Power-of-2 value

[Return values]

Result of e * 2^f operation

logf

Calculates the natural logarithm of a floating-point number.

[Format]

```
#include <mathf.h>
float logf (float f);
```

[Parameters]

f Floating-point number for which natural logarithm is to be computed

[Return values]

Normal: Natural logarithm of f
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if f is negative.
A range error occurs if f is 0.0.

log10f

Calculates the base-ten logarithm of a floating-point number.

[Format]

```
#include <mathf.h>
float log10f (float f);
```

[Parameters]

f Floating-point number for which base-ten logarithm is to be computed

[Return values]

Normal: Base-ten logarithm of f
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if f is negative.
A range error occurs if f is 0.0.

modff

Breaks a floating-point number into integral and fractional parts.

[Format]

```
#include <mathf.h>
float modff (float a, float *b);
```

[Parameters]

a Floating-point number to be broken into integral and fractional parts
b Pointer indicating storage area that stores integral part

[Return values]

Fractional part of **a**

powf

Calculates a power of a floating-point number.

[Format]

```
#include <mathf.h>
float powf (float x, float y);
```

[Parameters]

x Value to be raised to a power
y Power value

[Return values]

Normal: Value of x raised to the power y
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if x is 0.0 and y is 0.0 or less, or if x is negative and y is not an integer.

sqrtf

Calculates the positive square root of a floating-point number.

[Format]

```
#include <mathf.h>
float sqrtf (float f);
```

[Parameters]

f Floating-point number for which positive square root is to be computed

[Return values]

Normal: Positive square root of f
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if f is negative.

ceilf

Returns the smallest integral value not less than or equal to the given floating-point number.

[Format]

```
#include <mathf.h>
float ceilf (float f);
```

[Parameters]

f Floating-point number for which smallest integral value not less than that number is to be computed

[Return values]

Smallest integral value not less than or equal to f

[Remarks]

The ceilf function returns the smallest integral value not less than or equal to f, expressed as a float type value. Therefore, if f is negative, the value after truncation of the fractional part is returned.

fabsf

Calculates the absolute value of a floating-point number.

[Format]

```
#include <mathf.h>
float fabsf (float f);
```

[Parameters]

f Floating-point number for which absolute value is to be computed

[Return values]

Absolute value of f

floorf

Returns the largest integral value not greater than or equal to the given floating-point number.

[Format]

```
#include <mathf.h>
float floorf (float f);
```

[Parameters]

f Floating-point number for which largest integral value not greater than that number is to be computed

[Return values]

Largest integral value not greater than or equal to f

[Remarks]

The floorf function returns the largest integral value not greater than or equal to f, expressed as a float type value. Therefore, if f is negative, the value after rounding-up of the fractional part is returned.

fmodf

Calculates the remainder of a division of two floating-point numbers.

[Format]

```
#include <mathf.h>
float fmodf (float x, float y);
```

[Parameters]

x Dividend
y Divisor

[Return values]

When y is 0.0: x
When y is not 0.0: Remainder of division of x by y

[Remarks]

In the fmodf function, the relationship between parameters x and y and return value ret is as follows:
x = y * i + ret (where i is an integer)
The sign of return value ret is the same as the sign of x.
If the quotient of x/y cannot be represented, the value of the result is not guaranteed.

7.4.9 <setjmp.h>

Supports transfer of control between functions.
The following macros are implementation-defined.

Type	Definition Name	Description
Type (macro)	jmp_buf	Indicates the type name corresponding to a storage area for storing information that enables transfer of control between functions.
Function	setjmp	Saves the execution environment defined by **jmp_buf** of the currently executing function in the specified storage area.
	longjmp	Restores the function execution environment saved by the **setjmp** function, and transfers control to the program location at which the **setjmp** function was called.

The **setjmp** function saves the execution environment of the current function. The location in the program that called the **setjmp** function can subsequently be returned to by calling the **longjmp** function.

An example of how transfer of control between functions is supported using the **setjmp** and **longjmp** functions is shown below.

[Format]

```
1     #include <stdio.h>
2     #include <setjmp.h>
3     jmp_buf env;
4     void sub( );
5     void main( )
6     {
7
8         if (setjmp(env)!=0){
9             printf("return from longjmp\n");
10            exit(0);
11        }
12            sub( );
13    }
14
15    void sub( )
16    {
17        printf("subroutine is running \n");
18        longjmp(env, 1);
19    }
```

Explanation:

The **setjmp** function is called in line 8. At this time, the environment in which the **setjmp** function was called is saved in **jmp_buf** type variable **env**. The return value in this case is 0, and therefore function **sub** is called next.

The environment saved in variable **env** is restored by the **longjmp** function called within function **sub**. As a result, the program behaves just as if a return had been made from the **setjmp** function in line 8. However, the return value at this time is 1 specified by the second argument of the **longjmp** function. As a result, execution proceeds to line 9.

setjmp

Saves the execution environment of the currently executing function in the specified storage area.

[Format]

```
#include <setjmp.h>
long setjmp (jmp_buf env);
```

[Parameters]

env Pointer to storage area in which execution environment is to be saved

[Return values]

When **setjmp** function is called: 0
On return from **longjmp** function: Nonzero

[Remarks]

The execution environment saved by the **setjmp** function is used by the **longjmp** function. The return value is 0 when the function is called as the **setjmp** function, but the return value on return from the **longjmp** function is the value of the second parameter specified by the **longjmp** function.

If the **setjmp** function is called from a complex expression, part of the current execution environment, such as the intermediate result of expression evaluation, may be lost. The **setjmp** function should only be used in the form of a comparison between the result of the **setjmp** function and a constant expression, and should not be called within a complex expression.

Do not call the **setjmp** function indirectly using a pointer.

longjmp

Restores the function execution environment saved by the setjmp function, and transfers control to the program location at which the setjmp function was called.

[Format]

```
#include <setjmp.h>
void longjmp (jmp_buf env, long ret);
```

[Parameters]

env Pointer to storage area in which execution environment was saved
ret Return code to setjmp function

[Remarks]

From the storage area specified by the first parameter env, the longjmp function restores the function execution environment saved by the most recent invocation of the setjmp function in the same program, and transfers control to the program location at which that setjmp function was called. The value of the second parameter ret of the longjmp function is returned as the setjmp function return value. However, if ret is 0, the value 1 is returned to the setjmp function as a return value.

If the setjmp function has not been called, or if the function that called the setjmp function has already executed a return statement, the operation of the longjmp function is not guaranteed.

7.4.10 <stdarg.h>

Enables referencing of variable arguments for functions with such arguments.
The following macros are implementation-defined.

Type	Definition Name	Description
Type (macro)	va_list	Indicates the types of variables used in common by the **va_start**, **va_arg**, and **va_end** macros in order to reference variable arguments.
Function (macro)	va_start	Executes initialization processing for performing variable argument referencing.
	va_arg	Enables referencing of the argument following the argument currently being referenced for a function with variable arguments.
	va_end	Terminates referencing of the arguments of a function with variable arguments.
	va_copy *<-lang=c99>*	Copies variable arguments.

An example of a program using the macros defined by this standard include file is shown below.
[Format]

```
 1   #include <stdio.h>
 2   #include <stdarg.h>
 3
 4   extern void prlist(int count, ...);
 5
 6   void main( )
 7   {
 8       prlist(1, 1);
 9       prlist(3, 4, 5, 6);
10       prlist(5, 1, 2, 3, 4, 5);
11   }
12
13   void prlist(int count, ...)
14   {
15       va_list ap;
16       int i;
17
18       va_start(ap, count);
19       for(i=0; i<count; i++)
20           printf("%d", va_arg(ap, int));
21       putchar('\n');
22       va_end(ap);
23  }
```

Explanation:
This example implements function **prlist**, in which the number of data items to be output is specified in the first argument and that number of subsequent arguments are output.
In line 18, the variable argument reference is initialized by **va_start**. Each time an argument is output, the next argument is referenced by the **va_arg** macro (line 20). In the **va_arg** macro, the type name of the argument (in this case, **int** type) is specified in the second argument.
When argument referencing ends, the **va_end** macro is called (line 22).

va_start

Executes initialization processing for referencing variable arguments.

[Format]

```
#include <stdarg.h>
void va_start (va_list ap, parmN);
```

[Parameters]

ap Variable for accessing variable arguments
parmN Identifier of rightmost argument

[Remarks]

The va_start macro initializes ap for subsequent use by the va_arg and va_end macros.

The argument parmN is the identifier of the rightmost argument in the argument list in the external function definition (the one just before the , ...).

To reference variable unnamed arguments, the va_start macro call must be executed first of all.

va_arg

Allows a reference to the argument following the argument currently being referred to in the function with variable arguments.

[Format]

```
#include <stdarg.h>
type va_arg (va_list ap, type);
```

[Parameters]

ap Variable for accessing variable arguments
type Type of arguments to be accessed

[Return values]

Argument value

[Remarks]

Specify a variable of the va_list type initialized by the va_start macro as the first argument. The value of ap is updated each time va_arg is used, and, as a result, a sequence of variable arguments is returned by sequential calls of this macro.

Specify the type to refer to as the second argument type.

The ap argument must be the same as the ap initialized by va_start.

It will not be possible to refer to arguments correctly if argument type is set to a type of which size is changed by type conversion when it is used as a function argument, i.e., if char type, unsigned char type, short type, unsigned short type, or float type is specified as type. If such a type is specified, correct operation is not guaranteed.

va_end

Terminates referencing of the arguments of a function with variable arguments.

[Format]

```
#include <stdarg.h>
void va_end (va_list ap);
```

[Parameters]

ap Variable for referencing variable arguments

[Remarks]

The **ap** argument must be the same as the **ap** initialized by **va_start**. If the **va_end** macro is not called before the return from a function, the operation of that function is not guaranteed.

va_copy

Makes a copy of the argument currently being referenced for a function with variable arguments.

[Format]

```
#include <stdarg.h>
void va_copy (va_list dest, va_list src);
```

[Parameters]

dest Copy of variable for referencing variable arguments
src Variable for referencing variable arguments

[Remarks]

A copy is made of the second argument src which is one of the variable arguments that have been initialized by the va_start macro and used by the va_arg macro, and the copy is saved in the first argument dest.

The src argument must be the same as the src initialized by va_start.

The dest argument can be used as an argument that indicates the variable arguments in the subsequent va_arg macros.

7.4.11 <stdio.h>

Performs processing relating to input/output of stream input/output file.
The following constants (macros) are all implementation-defined.

Type	Definition Name	Description
Constant (macro)	FILE	Indicates a structure type that stores various control information including a pointer to the buffer, an error indicator, and an end-of-file indicator, which are required for stream input/output processing.
	_IOFBF	Indicates full buffering of input/output as the buffer area usage method.
	_IOLBF	Indicates line buffering of input/output as the buffer area usage method.
	_IONBF	Indicates non-buffering of input/output as the buffer area usage method.
	BUFSIZ	Indicates the buffer size required for input/output processing.
	EOF	Indicates end-of-file, that is, no more input from a file.
	L_tmpnam*	Indicates the size of an array large enough to store a string of a temporary file name generated by the **tmpnam** function.
	SEEK_CUR	Indicates a shift of the current file read/write position to an offset from the current position.
	SEEK_END	Indicates a shift of the current file read/write position to an offset from the end-of-file position.
	SEEK_SET	Indicates a shift of the current file read/write position to an offset from the beginning of the file.
	SYS_OPEN*	Indicates the number of files for which simultaneous opening is guaranteed by the implementation.
	TMP_MAX*	Indicates the maximum number of unique file names that shall be generated by the **tmpnam** function.
	stderr	Indicates the file pointer to the standard error file.
	stdin	Indicates the file pointer to the standard input file.
	stdout	Indicates the file pointer to the standard output file.

Type	Definition Name	Description
Function	fclose	Closes a stream input/output file.
	fflush	Outputs stream input/output file buffer contents to the file.
	fopen	Opens a stream input/output file under the specified file name.
	freopen	Closes a currently open stream input/output file and reopens a new file under the specified file name.
	setbuf	Defines and sets a stream input/output buffer area on the user program side.
	setvbuf	Defines and sets a stream input/output buffer area on the user program side.
	fprintf	Outputs data to a stream input/output file according to a format.
	vfprintf	Outputs a variable parameter list to the specified stream input/output file according to a format.
	printf	Converts data according to a format and outputs it to the standard output file (**stdout**).
	vprintf	Outputs a variable parameter list to the standard output file (**stdout**) according to a format.
	sprintf	Converts data according to a format and outputs it to the specified area.
	sscanf	Inputs data from the specified storage area and converts it according to a format.
	snprintf *<-lang=c99>*	Converts data according to a format and writes it to the specified array.
	vsnprintf *<-lang=c99>*	Equivalent to **snprintf** with the variable argument list replaced by **va_list**.
	vfscanf *<-lang=c99>*	Equivalent to **fscanf** with the variable argument list replaced by **va_list**.
	vscanf *<-lang=c99>*	Equivalent to **scanf** with the variable argument list replaced by **va_list**.
	vsscanf *<-lang=c99>*	Equivalent to **sscanf** with the variable argument list replaced by **va_list**.
	fscanf	Inputs data from a stream input/output file and converts it according to a format.
	scanf	Inputs data from the standard input file (**stdin**) and converts it according to a format.
	vsprintf	Outputs a variable parameter list to the specified area according to a format.
	fgetc	Inputs one character from a stream input/output file.
	fgets	Inputs a string from a stream input/output file.
	fputc	Outputs one character to a stream input/output file.
	fputs	Outputs a string to a stream input/output file.
	getc	(macro) Inputs one character from a stream input/output file.
	getchar	(macro) Inputs one character from the standard input file.
	gets	Inputs a string from the standard input file.
	putc	(macro) Outputs one character to a stream input/output file.
	putchar	(macro) Outputs one character to the standard output file.
	puts	Outputs a string to the standard output file.
	ungetc	Returns one character to a stream input/output file.

Type	Definition Name	Description
Function	fread	Inputs data from a stream input/output file to the specified storage area.
	fwrite	Outputs data from a storage area to a stream input/output file.
	fseek	Shifts the current read/write position in a stream input/output file.
	ftell	Obtains the current read/write position in a stream input/output file.
	rewind	Shifts the current read/write position in a stream input/output file to the beginning of the file.
	clearerr	Clears the error state of a stream input/output file.
	feof	Tests for the end of a stream input/output file.
	ferror	Tests for stream input/output file error state.
	perror	Outputs an error message corresponding to the error number to the standard error file (**stderr**).
Type	fpos_t	Indicates a type that can specify any position in a file.
Constant (macro)	FOPEN_MAX	Indicates the maximum number of files that can be opened simultaneously.
	FILENAME_MAX	Indicates the maximum length of a file name that can be held.

Note * These macros are not defined in this implementation.

Implementation-Defined Specifications

Item	Compiler Specifications
Whether the last line of the input text requires a new-line character indicating the end	Not specified. Depends on the low-level interface routine specifications.
Whether the space characters written immediately before the new-line character are read	
Number of null characters added to data written in the binary file	
Initial value of file position indicator in the append mode	
Whether file data is lost after output to a text file	
File buffering specifications	
Whether a file with file length 0 exists	
File name configuration rule	
Whether the same file is opened simultaneously	
Output data representation of the **%p** format conversion in the **fprintf** function	Hexadecimal representation.
Input data representation of the **%p** format conversion in the **fscanf** function. The meaning of conversion specifier '–' in the **fscanf** function	Hexadecimal representation. If '–' is not the first or last character or '–' does not follow '^', the range from the previous character to the following character is indicated.
Value of **errno** specified by the **fgetpos** or **ftell** function	The **fgetpos** function is not supported. The **errno** value for the **ftell** function is not specified. It depends on the low-level interface routine specifications.

Item	Compiler Specifications
Output format of messages generated by the **perror** function	See (a) below for the output message format.

(a) The output format of **perror** function is
<string>:<error message for the error number specified in error>

(b) Table 7.8 shows the format when displaying the floating-point infinity and not-a-number in **printf** and **fprintf** functions.

Table 7.8 Display Format of Infinity and Not-a-Number

Value	Display Format
Positive infinity	++++++
Negative infinity	------
Not-a-number	******

An example of a program that performs a series of input/output processing operations for a stream input/output file is shown in the following.

[Format]

```
1   #include <stdio.h>
2
3  void main( )
4  {
5      int c;
6      FILE *ifp, *ofp;
7
8      if ((ifp=fopen("INPUT.DAT","r"))==NULL){
9          fprintf(stderr,"cannot open input file\n");
10         exit(1);
11      }
12      if ((ofp=fopen("OUTPUT.DAT","w"))==NULL){
13          fprintf(stderr,"cannot open output file\n");
14          exit(1);
15      }
16      while ((c=getc(ifp))!=EOF)
17         putc(c, ofp);
18      fclose(ifp);
19      fclose(ofp);
20  }
```

Explanation:

This program copies the contents of file **INPUT.DAT** to file **OUTPUT.DAT**.

Input file **INPUT.DAT** is opened by the **fopen** function in line 8, and output file **OUTPUT.DAT** is opened by the **fopen** function in line 12. If opening fails, **NULL** is returned as the return value of the **fopen** function, an error message is output, and the program is terminated.

If the **fopen** function ends normally, the pointer to the data (**FILE** type) that stores information on the opened files is returned; these are set in variables **ifp** and **ofp**.

After successful opening, input/output is performed using these **FILE** type data.

When file processing ends, the files are closed with the **fclose** function.

fclose

Closes a stream input/output file.

[Format]

```
#include <stdio.h>
long fclose (FILE *fp);
```

[Parameters]

fp File pointer

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

The fclose function closes the stream input/output file indicated by file pointer fp.

If the output file of the stream input/output file is open and data that is not output remains in the buffer, that data is output to the file before it is closed.

If the input/output buffer was automatically allocated by the system, it is released.

fflush

Outputs the stream input/output file buffer contents to the file.

[Format]

```
#include <stdio.h>
long fflush (FILE *fp);
```

[Parameters]

fp File pointer

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

When the output file of the stream input/output file is open, the fflush function outputs the contents of the buffer that is not output for the stream input/output file specified by file pointer fp to the file. When the input file is open, the ungetc function specification is invalidated.

fopen

Opens a stream input/output file under the specified file name.

[Format]

```
#include <stdio.h>
FILE *fopen (const char *fname, const char *mode);
```

[Parameters]

fname Pointer to string indicating file name
mode Pointer to string indicating file access mode

[Return values]

Normal: File pointer indicating file information on opened file
Abnormal: NULL

[Remarks]

The fopen function opens the stream input/output file whose file name is the string pointed to by fname. If a file that does not exist is opened in write mode or append mode, a new file is created wherever possible. When an existing file is opened in write mode, writing processing is performed from the beginning of the file, and previously written file contents are erased.

When a file is opened in append mode, write processing is performed from the end-of-file position. When a file is opened in update mode, both input and output processing can be performed on the file. However, input cannot directly follow output without intervening execution of the fflush, fseek, or rewind function. Similarly, output cannot directly follow input without intervening execution of the fflush, fseek, or rewind function.

A string indicating the opening method may be added after the string indicating the file access mode.

freopen

Closes a currently open stream input/output file and reopens a new file under the specified file name.

[Format]

```
#include <stdio.h>
FILE *freopen (const char *fname,  const char *mode, FILE *fp);
```

[Parameters]

fname Pointer to string indicating new file name
mode Pointer to string indicating file access mode
fp File pointer to currently open stream input/output file

[Return values]

Normal: fp
Abnormal: NULL

[Remarks]

The freopen function first closes the stream input/output file indicated by file pointer fp (the following processing is carried out even if this close processing is unsuccessful). Next, the freopen function opens the file indicated by file name fname for stream input/output, reusing the FILE structure pointed to by fp.

The freopen function is useful when there is a limit on the number of files being opened at one time.

The freopen function normally returns the same value as fp, but returns NULL when an error occurs.

setbuf

Defines and sets a stream input/output buffer area by the user program.

[Format]

```
#include <stdio.h>
void setbuf (FILE *fp, char buf[BUFSIZ]);
```

[Parameters]

fp File pointer
buf Pointer to buffer area

[Remarks]

The setbuf function defines the storage area pointed to by buf so that it can be used as an input/output buffer area for the stream input/output file indicated by file pointer fp. As a result, input/output processing is performed using a buffer area of size BUFSIZ.

setvbuf

Defines and sets a stream input/output buffer area by the user program.

[Format]

```
#include <stdio.h>
long setvbuf (FILE *fp, char *buf, long type, size_t size);
```

[Parameters]

fp File pointer
buf Pointer to buffer area
type Buffer management method
size Size of buffer area

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

The setvbuf function defines the storage area pointed to by buf so that it can be used as an input/output buffer area for the stream input/output file indicated by file pointer fp.

There are three ways of using this buffer area, as follows:

(a) When _IOFBF is specified as type
Input/output is fully buffered.

(b) When _IOLBF is specified as type
Input/output is line buffered; that is, input/output data is fetched from the buffer area when a new-line character is written, when the buffer area is full, or when input is requested.

(c) When _IONBF is specified as type
Input/output is unbuffered. The setvbuf function usually returns 0. However, when an illegal value is specified for type or size, or when the request on how to use the buffer could not be accepted, a value other than 0 is returned.
The buffer area must not be released before the open stream input/output file is closed. In addition, the setvbuf function must be used between opening of the stream input/output file and execution of input/output processing.

fprintf

Outputs data to a stream input/output file according to the format.

[Format]

#include <stdio.h>
long fprintf (FILE *fp, const char *control[, arg]...);

[Parameters]

fp File pointer
control Pointer to string indicating format
arg,... List of data to be output according to format

[Return values]

Normal: Number of characters converted and output
Abnormal: Negative value

[Remarks]

The fprintf function converts and edits parameter arg according to the string that represents the format pointed to by control, and outputs the result to the stream input/output file indicated by file pointer fp.

The fprintf function returns the number of characters converted and output when the function is terminated successfully, or a negative value if an error occurs.

The format specifications are shown below.

Overview of Formats

The string that represents the format is made up of two kinds of string.

- Ordinary characters
  A character other than a conversion specification shown below is output unchanged.
- Conversion specifications
  A conversion specification is a string beginning with % that specifies the conversion method for the following parameter. The conversion specifications format conforms to the following rules:

%[Flag...] { [*] | [Field width] } { . { [*] | [Precision] } } [Parameter size specification] Conversion specifier

When there is no parameter to be actually output according to this conversion specification, the behavior is not guaranteed. In addition, when the number of parameters to be actually output is greater than the conversion specification, the excess parameters are ignored.

Description of Conversion Specifications

(a) Flags
Flags specify modifications to the data to be output, such as addition of a sign. The types of flag that can be specified and their meanings are shown in Table 7.9.

Table 7.9 Flag Types and Their Meanings

Type	Meaning
–	If the number of converted data characters is less than the field width, the data will be output left-justified within the field.
+	A plus or minus sign will be prefixed to the result of a signed conversion.

Type	Meaning
space	If the first character of a signed conversion result is not a sign, a space will be prefixed to the result. If the space and + flags are both specified, the space flag will be ignored.
#	The converted data is to be modified according to the conversion types described in Table 7.11. 1. For **c**, **d**, **i**, **s**, and **u** conversions This flag is ignored. 2. For **o** conversion The converted data is prefixed with 0. 3. For **x** or **X** conversion The converted data is prefixed with 0x (or 0X) 4. For **e**, **E**, **f**, **g**, and **G** conversions A decimal point is output even if the converted data has no fractional part. With **g** and **G** conversions, the 0 suffixed to the converted data are not removed.

(b) Field width
The number of characters in the converted data to be output is specified as a decimal number.
If the number of converted data characters is less than the field width, the data is prefixed with spaces up to the field width. (However, if '-' is specified as a flag, spaces are suffixed to the data.)
If the number of converted data characters exceeds the field width, the field width is extended to allow the converted result to be output.
If the field width specification begins with 0, the output data is prefixed with characters "0", not spaces.

(c) Precision
The precision of the converted data is specified according to the type of conversion, as described in Table 7.11.
The precision is specified in the form of a period (.) followed by a decimal integer. If the decimal integer is omitted, 0 is assumed to be specified.
If the specified precision is incompatible with the field width specification, the field width specification is ignored.
The precision specification has the following meanings according to the conversion type.

- For **d**, **i**, **o**, **u**, **x**, and **X** conversions
  The minimum number of digits in the converted data is specified.
- For **e**, **E**, and **f** conversions
  The number of digits after the decimal point in the converted data is specified.
- For **g** and **G** conversions
  The maximum number of significant digits in the converted data is specified.
- For **s** conversion
  The maximum number of printed digits is specified.

(d) Parameter size specification
For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions (see Table 7.10), the size (**short** type, **long** type, **long long** type, or **long double** type) of the data to be converted is specified. In other conversions, this specification is ignored. Table 7.10 shows the types of size specification and their meanings.

Table 7.10 Parameter Size Specification Types and Meanings

	Type	Meaning
1	hh	For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, specifies that the data to be converted is of **signed char** type or **unsigned char** type. For **n** conversion, specifies that the data to be converted is of pointer type to **signed char** type.
2	h	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the data to be converted is of **short** type or **unsigned short** type. For **n** conversion, specifies that the data to be converted is of pointer type to **short** type.
3	l	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the data to be converted is of **long** type, **unsigned long** type, or **double** type.
4	L	For **e**, **E**, **f**, **g**, and **G** conversions, specifies that the data to be converted is of **long double** type.

	Type	Meaning
5	ll	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the data to be converted is of **long long** type or **unsigned long long** type. For **n** conversion, specifies that the data to be converted is of pointer type to **long long** type.
6	j	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the data to be converted is of **intmax_t** type or **uintmax_t** type. For **n** conversion, specifies that the data to be converted is of pointer type to **size_t** type.
7	z	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the data to be converted is of **size_t** type or **signed integer** type corresponding to **size_t** type. For **n** conversion, specifies that the data to be converted is of pointer type to **size_t** type.
8	t	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the data to be converted is of **ptrdiff_t** type or **unsigned integer** type corresponding to **ptrdiff_t** type. For **n** conversion, specifies that the data to be converted is of pointer type to **ptrdiff_t** type.

(e) Conversion specifier

The format into which the data is to be converted is specified.

If the data to be converted is structure or array type, or is a pointer pointing to those types, the behavior is not guaranteed except when a character array is converted by **s** conversion or when a pointer is converted by **p** conversion. Table 7.11 shows the conversion specifier and conversion methods. If a letter which is not shown in this table is specified as the conversion specifier, the behavior is not guaranteed. The behavior, if a character that is not a letter is specified, depends on the compiler.

Table 7.11 Conversion Specifiers and Conversion Methods

	Conversion Specifier	Conversion Type	Conversion Method	Type Subject to Conversion	Notes Related to Precision
1	d	d conversion	**int** type data is converted to a signed decimal string. **d** conversion and **i** conversion have the same specification.	**int** type	The precision specification indicates the minimum number of characters output. If the number of converted data characters is less than the precision specification, the string is prefixed with zeros. If the precision is omitted, 1 is assumed. If conversion and output of data with a value of 0 is attempted with 0 specified as the precision, nothing will be output.
2	i	i conversion		**int** type	
3	o	o conversion	**int** type data is converted to an unsigned octal string.	**int** type	
4	u	u conversion	**int** type data is converted to an unsigned decimal string.	**int** type	
5	x	x conversion	**int** type data is converted to unsigned hexadecimal. a, b, c, d, e, and f are used as hexadecimal characters.	**int** type	
6	X	X conversion	**int** type data is converted to unsigned hexadecimal. A, B, C, D, E, and F are used as hexadecimal characters.	**int** type	

	Conversion Specifier	Conversion Type	Conversion Method	Type Subject to Conversion	Notes Related to Precision
7	f	f conversion	**double** type data is converted to a decimal string with the format [-] ddd.ddd.	**double** type	The precision specification indicates the number of digits after the decimal point. When there are characters after the decimal point, at least one digit is output before the decimal point. When the precision is omitted, 6 is assumed. When 0 is specified as the precision, the decimal point and subsequent characters are not output. The output data is rounded.
8	F	F conversion		**double** type	
9	e	e conversion	**double** type data is converted to a decimal string with the format [-] d.ddde±dd. At least two digits are output as the exponent.	**double** type	The precision specification indicates the number of digits after the decimal point. The format is such that one digit is output before the decimal point in the converted characters, and a number of digits equal to the precision are output after the decimal point. When the precision is omitted, 6 is assumed. When 0 is specified as the precision, characters after the decimal point are not output. The output data is rounded.
10	E	E conversion	**double** type data is converted to a decimal string with the format [-] d.dddE±dd. At least two digits are output as the exponent.	**double** type	
11	g	g conversion (or G conversion)	Whether **f** conversion format output or **e** conversion (or **E** conversion) format output is performed is determined by the value to be converted and the precision value that specifies the number of significant digits. Then **double** type data is output. If the exponent of the converted data is less than –4, or larger than the precision that indicates the number of significant digits, conversion to **e** (or **E**) format is performed.	**double** type	The precision specification indicates the maximum number of significant digits in the converted data.
12	G			**double** type	

	Conversion Specifier	Conversion Type	Conversion Method	Type Subject to Conversion	Notes Related to Precision
13	a	a conversion	**double** type data is converted to a hexadecimal string with the [-]0xh.hhhhp ± d format. At least one digit is output as the exponent.	**double** type	The precision specification indicates the number of digits after the decimal point. The format is such that one digit is output before the decimal point in the converted characters, and a number of digits equal to the precision are output after the decimal point. When the precision is omitted, a precision sufficient for representing an accurate value is assumed. When 0 is specified as the precision, characters after the decimal point are not output. The output data is rounded.
14	A	A conversion	**double** type data is converted to a hexadecimal string with the [-]0Xh.hhhhP ± d format. At least one digit is output as the exponent.	**double** type	
15	c	c conversion	**int** type data is converted to **unsigned char** data, with conversion to the character corresponding to that data.	**int** type	The precision specification is invalid.
16	s	s conversion	The string pointed to by pointer to **char** type are output up to the null character indicating the end of the string or up to the number of characters specified by the precision. (Null characters are not output. Space, horizontal tab, and new-line characters are not included in the converted string.)	Pointer to **char** type	The precision specification indicates the number of characters to be output. If the precision is omitted, characters are output up to, but not including, the null character in the string pointed to by the data. (Null characters are not output. Space, horizontal tab, and new-line characters are not included in the converted string.)
17	p	p conversion	Assuming data as a pointer, conversion is performed to a string of compiler-defined printable characters.	Pointer to **void** type	The precision specification is invalid.
18	n	No conversion is performed.	Data is regarded as a pointer to **int** type, and the number of characters output so far is set in the storage area pointed to by that data.	Pointer to **int** type	
19	%	No conversion is performed.	% is output.	None	

(f) * specification for field width or precision
* can be specified as the field width or precision specification value. In this case, the value of the parameter corresponding to the conversion specification is used as the field width or precision specification value. When this parameter has a negative field width, it is interpreted as flag '–' and a positive field width. When the parameter has a negative precision, the precision is interpreted as being omitted.

vfprintf

Outputs a variable parameter list to the specified stream input/output file according to a format.

[Format]

```
#include <stdarg.h>
#include <stdio.h>
long vfprintf (FILE *fp, const char *control, va_list arg);
```

[Parameters]

fp File pointer
control Pointer to string indicating format
arg Parameter list

[Return values]

Normal: Number of characters converted and output
Abnormal: Negative value

[Remarks]

The **vfprintf** function sequentially converts and edits a variable parameter list according to the string that represents the format pointed to by **control**, and outputs the result to the stream input/output file indicated by **fp**.

The **vfprintf** function returns the number of data items converted and output, or a negative value when an error occurs.

Within the **vfprintf** function, the **va_end** macro is not invoked.

For details of the format specifications, see the description of the **fprintf** function.

Parameter **arg**, indicating the parameter list, must be initialized beforehand by the **va_start** macro (and the succeeding **va_arg** macro).

printf

Converts data according to a format and outputs it to the standard output file (stdout).

[Format]

```
#include <stdio.h>
long printf (const char *control[, arg]...);
```

[Parameters]

control Pointer to string indicating format
arg,... Data to be output according to format

[Return values]

Normal: Number of characters converted and output
Abnormal: Negative value

[Remarks]

The **printf** function converts and edits parameter arg according to the string that represents the format pointed to by control, and outputs the result to the standard output file (stdout).

For details of the format specifications, see the description of the **fprintf** function.

vprintf

Outputs a variable parameter list to the standard output file (stdout) according to a format.

[Format]

```
#include <stdarg.h>
#include <stdio.h>
long vprintf (const char *control, va_list arg);
```

[Parameters]

control Pointer to string indicating format
arg Parameter list

[Return values]

Normal: Number of characters converted and output
Abnormal: Negative value

[Remarks]

The **vprintf** function sequentially converts and edits a variable parameter list according to the string that represents the format pointed to by **control**, and outputs the result to the standard output file.

The **vprintf** function returns the number of data items converted and output, or a negative value when an error occurs.

Within the **vprintf** function, the **va_end** macro is not invoked.

For details of the format specifications, see the description of the **fprintf** function.

Parameter **arg**, indicating the parameter list, must be initialized beforehand by the **va_start** macro (and the succeeding **va_arg** macro).

sprintf

Converts data according to a format and outputs it to the specified area.

[Format]

```
#include <stdio.h>
long sprintf (char *s, const char *control[, arg...]);
```

[Parameters]

s Pointer to storage area to which data is to be output
control Pointer to string indicating format
arg,... Data to be output according to format

[Return values]

Number of characters converted

[Remarks]

The **sprintf** function converts and edits parameter **arg** according to the string that represents the format pointed to by **control**, and outputs the result to the storage area pointed to by **s**.

A null character is appended at the end of the converted and output string. This null character is not included in the return value (number of characters output).

For details of the format specifications, see the description of the **fprintf** function.

sscanf

Inputs data from the specified storage area and converts it according to a format.

[Format]

```
#include <stdio.h>
long sscanf (const char *s, const char *control[, ptr...]);
```

[Parameters]

s Storage area containing data to be input
control Pointer to string indicating format
ptr,... Pointer to storage area that stores input and converted data

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: **EOF**

[Remarks]

The **sscanf** function inputs data from the storage area pointed to by **s**, converts and edits it according to the string that represents the format pointed to by **control**, and stores the result in the storage area pointed to by **ptr**.

The **sscanf** function returns the number of data items successfully input and converted. **EOF** is returned when the input data ends before the first conversion.

For details of the format specifications, see the description of the **fscanf** function.

snprintf

Converts data according to a format and outputs it to the specified area.

[Format]

```
#include <stdio.h>
long snprintf(char *restrict s, size_t n, const char *restrict control [, arg]...);
```

[Parameters]

s Pointer to storage area to which data is to be output
n Number of characters to be output
control Pointer to string indicating format
arg,... Data to be output according to format

[Return values]

Number of characters converted

[Remarks]

The snprintf function converts and edits parameter arg according to the format-representing string pointed to by control, and outputs the result to the storage area pointed to by s.

A null character is appended at the end of the converted and output string. This null character is not included in the return value (number of characters output). For details of the format specifications, see the description of the fprintf function.

vsnprintf

Converts data according to a format and outputs it to the specified area.

[Format]

```
#include <stdarg.h>
#include <stdio.h>
long vsnprintf(char *restrict s, size_t n, const char *restrict control, va_list arg);
```

[Parameters]

s Pointer to storage area to which data is to be output
n Number of characters to be output
control Pointer to string indicating format
arg Parameter list

[Return values]

Number of characters converted

[Remarks]

The **vsnprintf** function is equivalent to **snprintf** with **arg** specified instead of the variable parameters.
Initialize **arg** through the **va_start** macro before calling the **vsnprintf** function.
The **vsnprintf** function does not call the **va_end** macro.

vfscanf

Inputs data from a stream input/output file and converts it according to a format.

[Format]

```
#include <stdarg.h>
#include <stdio.h>
long vfscanf(FILE *restrict fp, const char *restrict control, va_list arg);
```

[Parameters]

fp File pointer
control Pointer to wide string indicating format
arg Parameter list

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: Input data ends before input data conversion is performed: **EOF**

[Remarks]

The **vfscanf** function is equivalent to **fscanf** with **arg** specified instead of the variable parameter list.
Initialize **arg** through the **va_start** macro before calling the **vfscanf** function.
The **vfscanf** function does not call the **va_end** macro.

vscanf

Inputs data from the specified storage area and converts it according to a format.

[Format]

```
#include <stdarg.h>
#include <stdio.h>
long vscanf(const char *restrict control, va_list arg);
```

[Parameters]

control Pointer to string indicating format
arg Parameter list

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: Input data ends before input data conversion is performed: **EOF**

[Remarks]

The **vscanf** function is equivalent to **scanf** with **arg** specified instead of the variable parameters.
Initialize **arg** through the **va_start** macro before calling the **vscanf** function.
The **vscanf** function does not call the **va_end** macro.

vsscanf

Inputs data from the specified storage area and converts it according to a format.

[Format]

```
#include <stdarg.h>
#include <stdio.h>
long vsscanf(const char *restrict s, const char *restrict control, va_list arg);
```

[Parameters]

s Storage area containing data to be input
control Pointer to string indicating format
arg Parameter list

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: nput data ends before input data conversion is performed: **EOF**

[Remarks]

The **vsscanf** function is equivalent to **sscanf** with **arg** specified instead of the variable parameters.
Initialize **arg** through the **va_start** macro before calling the **vsscanf** function.
The **vsscanf** function does not call the **va_end** macro.

fscanf

Inputs data from a stream input/output file and converts it according to a format.

[Format]

#include <stdio.h>
long fscanf (FILE *fp, const char *control[, ptr]...);

[Parameters]

fp File pointer
control Pointer to string indicating format
ptr,... Pointer to storage area that stores input data

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: Input data ends before input data conversion is performed: **EOF**

[Remarks]

The **fscanf** function inputs data from the stream input/output file indicated by file pointer **fp**, converts and edits it according to the string that represents the format pointed to by **control**, and stores the result in the storage area pointed to by **ptr**.

The format specifications for inputting data are shown below.

Overview of Formats
The string that represents the format is made up of the following three kinds of string.

- Space characters
  If a space (' '), horizontal tab ('\t'), or new-line character ('\n') is specified, processing is performed to skip to the next non-white-space character in the input data.
- Ordinary characters
  If a character that is neither one of the space characters listed above nor % is specified, one input data character is input. The input character must match a character specified in the string that represents the format.
- Conversion specification
  A conversion specification is a string beginning with % that specifies the method of converting the input data and storing it in the area pointed to by the following parameter. The conversion specification format conforms to the following rules:
  % [*] [Field width] [Converted data size] Conversion specifier
  If there is no pointer to the storage area that stores input data corresponding to the conversion specification in the format, the behavior is not guaranteed. In addition, when a pointer to a storage area that stores input data remains though the format is exhausted, that pointer is ignored.

Description of Conversion Specifications

- * specification
  Suppresses storage of the input data in the storage area pointed to by the parameter.
- Field width
  The maximum number of characters in the data to be input is specified as a decimal number.
- Converted data size
  For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, and **f** conversions (see Table 7.13), the size (**short** type, **long** type, **long long** type, or **long double** type) of the converted data is specified. In other conversions, this specification is ignored. Table 7.12 shows the types of size specification and their meanings.

Table 7.12 Converted Data Size Specification Types and Meanings

	Type	Meaning
1	h	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the converted data is of **short** type.
2	l	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the converted data is of **long** type. For **e**, **E**, and **f** conversions, specifies that the converted data is of **double** type.
3	L	For **e**, **E**, and **f** conversions, specifies that the converted data is of **long double** type.
4	ll	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the converted data is of **long long** type.

- Conversion specifier
  The input data is converted according to the type of conversion specified by the conversion specifier. However, processing is terminated when a white-space character is read, when a character for which conversion is not permitted is read, or when the specified field width has been exceeded.

Table 7.13 Conversion Specifiers and Conversion Methods

	Conversion Specifier	Conversion Type	Conversion Method	Data Type Subject to Conversion
1	d	d conversion	A decimal string is converted to integer type data.	Integer type
2	i	i conversion	A decimal string with a sign prefixed, or a decimal string with u (U) or l (L) suffixed is converted to integer type data. A string beginning with 0x (or 0X) is interpreted as hexadecimal, and the string is converted to **int** type data. A string beginning with 0 is interpreted as octal, and the string is converted to **int** type data.	Integer type
3	o	o conversion	An octal string is converted to integer type data.	Integer type
4	u	u conversion	An unsigned decimal string is converted to integer type data.	Integer type
5	x	x conversion	A hexadecimal string is converted to integer type data. There is no difference in meaning between **x** conversion and **X** conversion.	Integer type
6	X	X conversion		
7	s	s conversion	Characters are converted as a single string until a space, horizontal tab, or new-line character is read. A null character is appended at the end of the string. (The string in which the converted data is set must be large enough to include the null character.)	Character type
8	c	c conversion	One character is input. The input character is not skipped even if it is a white-space character. To read only non-white-space characters, specify **%1s**. If the field width is specified, the number of characters equivalent to that specification are read. In this case, therefore, the storage area that stores the converted data needs the specified size.	**char** type

	Conversion Specifier	Conversion Type	Conversion Method	Data Type Subject to Conversion
9	e	e conversion	A string indicating a floating-point number is converted to floating-point type data. There is no difference in meaning between the **e** conversion and **E** conversion, or between the **g** conversion and **G** conversion. The input format is a floating-point number that can be represented by the **strtod** function.	Floating-point type
10	E	E conversion		
11	f	f conversion		
12	g	g conversion		
13	G	G conversion		
14	p	p conversion	A string converted by **p** conversion of the **fprintf** function is converted to pointer type data.	Pointer to **void** type
15	n	No conversion is performed.	Data input is not performed; the number of data characters input so far is set.	Integer type
16	[	[conversion	A set of characters is specified after [, followed by]. This character set defines a set of characters comprising a string. If the first character of the character set is not a circumflex (^), the input data is input as a single string until a character not in this character set is first read. If the first character is ^, the input data is input as a single string until a character which is in the character set following the ^ is first read. A null character is automatically appended at the end of the input string. (The string in which the converted data is set must be large enough to include the null character.)	Character type
17	%	No conversion is performed.	% is read.	None

If the conversion specifier is a letter not shown in Table 7.13, the behavior is not guaranteed. For the other characters, the behavior is implementation-defined.

scanf

Inputs data from the standard input file (stdin) and converts it according to a format.

[Format]

```
#include <stdio.h>
long scanf (const char *control[, ptr...]);
```

[Parameters]

control Pointer to string indicating format
ptr,... Pointer to storage area that stores input and converted data

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: **EOF**

[Remarks]

The **scanf** function inputs data from the standard input file (**stdin**), converts and edits it according to the string that represents the format pointed to by **control**, and stores the result in the storage area pointed to by **ptr**.

The **scanf** function returns the number of data items successfully input and converted as the return value. **EOF** is returned if the standard input file ends before the first conversion.

For details of the format specifications, see the description of the **fscanf** function.

For **%e** conversion, specify **l** for **double** type, and specify **L** for **long double** type. The default type is **float**.

vsprintf

Outputs a variable parameter list to the specified storage area according to a format.

[Format]

```
#include <stdarg.h>
#include <stdio.h>
long vsprintf (char *s, const char *control, va_list arg);
```

[Parameters]

s Pointer to storage area to which data is to be output
control Pointer to string indicating format
arg Parameter list

[Return values]

Normal: Number of characters converted
Abnormal: Negative value

[Remarks]

The **vsprintf** function sequentially converts and edits a variable parameter list according to the string that represents the format pointed to by **control**, and outputs the result to the storage area pointed to by **s**.

A null character is appended at the end of the converted and output string. This null character is not included in the return value (number of characters output).

For details of the format specifications, see the description of the **fprintf** function.

Parameter **arg**, indicating the parameter list, must be initialized beforehand by the **va_start** macro (and the succeeding **va_arg** macro).

fgetc

Inputs one character from a stream input/output file.

[Format]

```
#include <stdio.h>
long fgetc (FILE *fp);
```

[Parameters]

fp File pointer

[Return values]

Normal: End-of-file: **EOF**
Otherwise: Input character
Abnormal: **EOF**

[Remarks]

When a read error occurs, the error indicator for that file is set.

The **fgetc** function inputs one character from the stream input/output file indicated by file pointer **fp**.

The **fgetc** function normally returns the input character, but returns **EOF** at end-of-file or when an error occurs. At end-of-file, the end-of-file indicator for that file is set.

fgets

Inputs a string from a stream input/output file.

[Format]

#include <stdio.h>
char *fgets (char *s, long n, FILE *fp);

[Parameters]

s Pointer to storage area to which string is input
n Number of bytes of storage area to which string is input
fp File pointer

[Return values]

Normal: End-of-file: **NULL**
Otherwise: **s**
Abnormal: **NULL**

[Remarks]

The **fgets** function inputs a string from the stream input/output file indicated by file pointer **fp** to the storage area pointed to by **s**.

The **fgets** function performs input up to the (n–1)th character or a new-line character, or until end-of-file, and appends a null character at the end of the input string.

The **fgets** function normally returns **s**, the pointer to the storage area to which the string is input, but returns **NULL** at end-of-file or if an error occurs.

The contents of the storage area pointed to by **s** do not change at end-of-file, but are not guaranteed when an error occurs.

fputc

Outputs one character to a stream input/output file.

[Format]

```
#include <stdio.h>
long fputc (long c, FILE *fp);
```

[Parameters]

c Character to be output
fp File pointer

[Return values]

Normal: Output character
Abnormal: **EOF**

[Remarks]

When a write error occurs, the error indicator for that file is set.
The **fputc** function outputs character **c** to the stream input/output file indicated by file pointer **fp**.
The **fputc** function normally returns **c**, the output character, but returns **EOF** when an error occurs.

fputs

Outputs a string to a stream input/output file.

[Format]

```
#include <stdio.h>
long fputs (const char *s, FILE *fp);
```

[Parameters]

s Pointer to string to be output
fp File pointer

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

The **fputs** function outputs the string pointed to by **s** up to the character preceding the null character to the stream input/output file indicated by file pointer **fp**. The null character indicating the end of the string is not output.

The **fputs** function normally returns zero, but returns nonzero when an error occurs.

getc

Inputs one character from a stream input/output file.

[Format]

```
#include <stdio.h>
long getc (FILE *fp);
```

[Parameters]

fp File pointer

[Return values]

Normal: End-of-file: **EOF**
Otherwise: Input character
Abnormal: **EOF**

[Remarks]

When a read error occurs, the error indicator for that file is set.

The **getc** function inputs one character from the stream input/output file indicated by file pointer **fp**.

The **getc** function normally returns the input character, but returns **EOF** at end-of-file or when an error occurs. At end-of-file, the end-of-file indicator for that file is set.

getchar

Inputs one character from the standard input file (stdin).

[Format]

```
#include <stdio.h>
long getchar (void);
```

[Return values]

Normal: End-of-file: **EOF**
Otherwise: Input character
Abnormal: **EOF**

[Remarks]

When a read error occurs, the error indicator for that file is set.
The **getchar** function inputs one character from the standard input file (**stdin**).
The **getchar** function normally returns the input character, but returns **EOF** at end-of-file or when an error occurs. At end-of-file, the end-of-file indicator for that file is set.

gets

Inputs a string from the standard input file (stdin).

[Format]

```
#include <stdio.h>
char *gets (char *s);
```

[Parameters]

s Pointer to storage area to which string is input

[Return values]

Normal: End-of-file: **NULL**
Otherwise: **s**
Abnormal: **NULL**

[Remarks]

The **gets** function inputs a string from the standard input file (**stdin**) to the storage area starting at **s**.

The **gets** function inputs characters up to end-of-file or until a new-line character is input, and appends a null character instead of a new-line character.

The **gets** function normally returns **s**, the pointer to the storage area to which the string is input, but returns **NULL** at the end of the standard input file or when an error occurs.

The contents of the storage area pointed to by **s** do not change at the end of the standard input file, but are not guaranteed when an error occurs.

putc

Outputs one character to a stream input/output file.

[Format]

```
#include <stdio.h>
long putc (long c, FILE *fp);
```

[Parameters]

c Character to be output
fp File pointer

[Return values]

Normal: Output character
Abnormal: **EOF**

[Remarks]

When a write error occurs, the error indicator for that file is set.
The **putc** function outputs character **c** to the stream input/output file indicated by file pointer **fp**.
The **putc** function normally returns **c**, the output character, but returns **EOF** when an error occurs.

putchar

Outputs one character to the standard output file (stdout).

[Format]

```
#include <stdio.h>
long putchar (long c);
```

[Parameters]

c Character to be output

[Return values]

Normal: Output character
Abnormal: **EOF**

[Remarks]

When a write error occurs, the error indicator for that file is set.
The **putchar** function outputs character **c** to the standard output file (**stdout**).
The **putchar** function normally returns **c**, the output character, but returns **EOF** when an error occurs.

puts

Outputs a string to the standard output file (stdout).

[Format]

```
#include <stdio.h>
long puts (const char *s);
```

[Parameters]

s Pointer to string to be output

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

The **puts** function outputs the string pointed to by **s** to the standard output file (**stdout**). The null character indicating the end of the string is not output, but a new-line character is output instead.

The **puts** function normally returns zero, but returns nonzero when an error occurs.

ungetc

Returns one character to a stream input/output file.

[Format]

```
#include <stdio.h>
long ungetc (long c, FILE *fp);
```

[Parameters]

c Character to be returned
fp File pointer

[Return values]

Normal: Returned character
Abnormal: **EOF**

[Remarks]

The **ungetc** function returns character **c** to the stream input/output file indicated by file pointer **fp**. Unless the **fflush**, **fseek**, or **rewind** function is called, this returned character will be the next input data.

The **ungetc** function normally returns **c**, which is the returned character, but returns **EOF** when an error occurs.

The behavior is not guaranteed when the **ungetc** function is called more than once without intervening **fflush**, **fseek**, or **rewind** function execution. When the **ungetc** function is executed, the current file position indicator for that file is moved back one position; however, when this file position indicator has already been positioned at the beginning of the file, its value is not guaranteed.

fread

Inputs data from a stream input/output file to the specified storage area.

[Format]

```
#include <stdio.h>
size_t fread (void *ptr, size_t size, size_t n, FILE *fp);
```

[Parameters]

ptr Pointer to storage area to which data is input
size Number of bytes in one member
n Number of members to be input
fp File pointer

[Return values]

When **size** or **n** is 0: 0
When **size** and **n** are both nonzero: Number of successfully input members

[Remarks]

The **fread** function inputs **n** members whose size is specified by **size**, from the stream input/output file indicated by file pointer **fp**, into the storage area pointed to by **ptr**. The file position indicator for the file is advanced by the number of bytes input.

The **fread** function returns the number of members successfully input, which is normally the same as the value of **n**. However, at end-of-file or when an error occurs, the number of members successfully input so far is returned, and then the return value will be less than **n**. The **ferror** and **feof** functions should be used to distinguish between end-of-file and error occurrence.

When the value of **size** or **n** is zero, zero is returned as the return value and the contents of the storage area pointed to by **ptr** do not change. When an error occurs or when only a part of the members can be input, the file position indicator is not guaranteed.

fwrite

Outputs data from a memory area to a stream input/output file.

[Format]

```
#include <stdio.h>
size_t fwrite (const void *ptr, size_t size, size_t n, FILE *fp);
```

[Parameters]

ptr Pointer to storage area storing data to be output
size Number of bytes in one member
n Number of members to be output
fp File pointer

[Return values]

Number of successfully output members

[Remarks]

The **fwrite** function outputs **n** members whose size is specified by **size**, from the storage area pointed to by **ptr**, to the stream input/output file indicated by file pointer **fp**. The file position indicator for the file is advanced by the number of bytes output.

The **fwrite** function returns the number of members successfully output, which is normally the same as the value of **n**. However, when an error occurs, the number of members successfully output so far is returned, and then the return value will be less than **n**.

When an error occurs, the file position indicator is not guaranteed.

fseek

Shifts the current read/write position in a stream input/output file.

[Format]

```
#include <stdio.h>
long fseek (FILE *fp, long offset, long type);
```

[Parameters]

fp File pointer
offset Offset from position specified by type of offset
type Type of offset

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

The **fseek** function shifts the current read/write position in the stream input/output file indicated by file pointer **fp** by **offset** bytes from the position specified by **type** (the type of offset).
The types of offset are shown in Table 7.14.
The **fseek** function normally returns zero, but returns nonzero in response to an invalid request.

Table 7.14 Types of Offset

Offset Type	Meaning
SEEK_SET	Shifts to a position which is located **offset** bytes away from the beginning of the file. The value specified by **offset** must be zero or positive.
SEEK_CUR	Shifts to a position which is located **offset** bytes away from the current position in the file. The shift is toward the end of the file if the value specified by **offset** is positive, and toward the beginning of the file if negative.
SEEK_END	Shifts to a position which is located **offset** bytes forward from end-of-file. The value specified by **offset** must be zero or negative.

For a text file, the type of offset must be **SEEK_SET** and **offset** must be zero or the value returned by the **ftell** function for that file. Note also that calling the **fseek** function cancels the effect of the **ungetc** function.

ftell

Obtains the current read/write position in a stream input/output file.

[Format]

```
#include <stdio.h>
long ftell (FILE *fp);
```

[Parameters]

fp File pointer

[Return values]

Current file position indicator position (text file)
Number of bytes from beginning of file to current position (binary file)

[Remarks]

The **ftell** function obtains the current read/write position in the stream input/output file indicated by file pointer **fp**.

For a binary file, the **ftell** function returns the number of bytes from the beginning of the file to the current position. For a text file, it returns, as the position of the file position indicator, an implementation-defined value that can be used by the **fseek** function.

When the **ftell** function is used twice for a text file, the difference in the return values will not necessarily represent the actual distance in the file.

rewind

Shifts the current read/write position in a stream input/output file to the beginning of the file.

[Format]

```
#include <stdio.h>
void rewind (FILE *fp);
```

[Parameters]

fp File pointer

[Remarks]

The **rewind** function shifts the current read/write position in the stream input/output file indicated by file pointer **fp**, to the beginning of the file.

The **rewind** function clears the end-of-file indicator and error indicator for the file.

Note that calling the **rewind** function cancels the effect of the **ungetc** function.

clearerr

Clears the error state of a stream input/output file.

[Format]

```
#include <stdio.h>
void clearerr (FILE *fp);
```

[Parameters]

fp File pointer

[Remarks]

The **clearerr** function clears the error indicator and end-of-file indicator for the stream input/output file indicated by file pointer **fp**.

feof

Tests for the end of a stream input/output file.

[Format]

```
#include <stdio.h>
long feof (FILE *fp);
```

[Parameters]

fp File pointer

[Return values]

End-of-file: Nonzero
Otherwise: 0

[Remarks]

The **feof** function tests for the end of the stream input/output file indicated by file pointer **fp**.

The **feof** function tests the end-of-file indicator for the specified stream input/output file, and if the indicator is set, returns nonzero to indicate that the file is at its end. If the end-of-file indicator is not set, the **feof** function returns zero to show that the file is not yet at its end.

ferror

Tests for stream input/output file error state.

[Format]

```
#include <stdio.h>
long ferror (FILE *fp);
```

[Parameters]

fp File pointer

[Return values]

If file is in error state: Nonzero
Otherwise: 0

[Remarks]

The **ferror** function tests whether the stream input/output file indicated by file pointer **fp** is in the error state.

The **ferror** function tests the error indicator for the specified stream input/output file, and if the indicator is set, returns nonzero to show that the file is in the error state. If the error indicator is not set, the **ferror** function returns zero to show that the file is not in the error state.

perror

Outputs an error message corresponding to the error number to the standard error file (stderr).

[Format]

```
#include <stdio.h>
void perror (const char *s)
```

[Parameters]

s Pointer to error message

[Remarks]

The **perror** function maps **errno** to the error message indicated by **s**, and outputs the message to the standard error file (**stderr**).

If **s** is not **NULL** and the string pointed to by **s** is not a null character, the output format is as follows: the string pointed to by **s** followed by a colon and space, then the implementation-defined error message, and finally a new-line character.

7.4.12 <stdlib.h>

Defines standard functions for standard processing of C programs.
The following macros are implementation-defined.

Type	Definition Name	Description
Type (macro)	onexit_t	Indicates the type returned by the function registered by the **onexit** function and the type of the value returned by the **onexit** function.
	div_t	Indicates the type of structure of the value returned by the **div** function.
	ldiv_t	Indicates the type of structure of the value returned by the **ldiv** function.
	lldiv_t	Indicates the type of structure of the value returned by the **lldiv** function.
Constant (macro)	RAND_MAX	Indicates the maximum value of pseudo-random integers generated by the **rand** function.
	EXIT_SUCCESS	Indicates the successfully completed state.
Function	atof	Converts a number-representing string to a **double** type floating-point number.
	atoi	Converts a decimal-representing string to an **int** type integer.
	atol	Converts a decimal-representing string to a **long** type integer.
	atoll	Converts a decimal-representing string to a **long long** type integer.
	strtod	Converts a number-representing string to a **double** type floating-point number.
	strtof	Converts a number-representing string to a **float** type floating-point number.
	strtold	Converts a number-representing string to a **long double** type floating-point number.
	strtol	Converts a number-representing string to a **long** type integer.
	strtoul	Converts a number-representing string to an **unsigned long** type integer.
	strtoll	Converts a number-representing string to a **long long** type integer.
	strtoull	Converts a number-representing string to an **unsigned long long** type integer.
	rand	Generates pseudo-random integers from 0 to **RAND_MAX**.
	srand	Sets an initial value of the pseudo-random number sequence generated by the **rand** function.
	calloc	Allocates a storage area and clears all bits in the allocated storage area to 0.
	free	Releases specified storage area.
	malloc	Allocates a storage area.
	realloc	Changes the size of storage area to a specified value.
	bsearch	Performs binary search.
	qsort	Performs sorting.
	abs	Calculates the absolute value of an **int** type integer.

Type	Definition Name	Description
Function	div	Carries out division of **int** type integers and obtains the quotient and remainder.
	labs	Calculates the absolute value of a **long** type integer.
	ldiv	Carries out division of **long** type integers and obtains the quotient and remainder.
	llabs	Calculates the absolute value of a **long long** type integer.
	lldiv	Carries out division of **long long** type integers and obtains the quotient and remainder.

Implementation-Defined Specifications

Item	Compiler Specifications
calloc, **malloc**, or **realloc** function operation when the size is 0.	**NULL** is returned.

atof

Converts a number-representing string to a double type floating-point number.

[Format]

```
#include <stdlib.h>
double atof (const char *nptr);
```

[Parameters]

nptr Pointer to a number-representing string to be converted

[Return values]

Converted data as a **double** type floating-point number

[Remarks]

If the converted result overflows or underflows, **errno** is set.
Data is converted up to the first character that does not fit the floating-point data type.
The **atof** function does not guarantee the return value if an error such as an overflow occurs. When you want to acquire the guaranteed return value, use the **strtod** function.

atoi

Converts a decimal-representing string to an int type integer.

[Format]

```
#include <stdlib.h>
long atoi (const char *nptr);
```

[Parameters]

nptr Pointer to a number-representing string to be converted

[Return values]

Converted data as an int type integer

[Remarks]

If the converted result overflows, **errno** is set.

Data is converted up to the first character that does not fit the decimal data type.

The **atoi** function does not guarantee the return value if an error such as an overflow occurs. When you want to acquire the guaranteed return value, use the **strtol** function.

atol

Converts a decimal-representing string to a long type integer.

[Format]

```
#include <stdlib.h>
long atol (const char *nptr);
```

[Parameters]

nptr Pointer to a number-representing string to be converted

[Return values]

Converted data as a long type integer

[Remarks]

If the converted result overflows, **errno** is set.
Data is converted up to the first character that does not fit the decimal data type.
The **atol** function does not guarantee the return value if an error such as an overflow occurs. When you want to acquire the guaranteed return value, use the **strtol** function.

atoll

Converts a decimal-representing string to a **long long** type integer.

[Format]

```
#include <stdlib.h>
long long atoll (const char *nptr);
```

[Parameters]

nptr Pointer to a number-representing string to be converted

[Return values]

Converted data as a **long long** type integer

[Remarks]

If the converted result overflows, **errno** is set.

Data is converted up to the first character that does not fit the decimal data type.

The **atoll** function does not guarantee the return value if an error such as an overflow occurs. When you want to acquire the guaranteed return value, use the **strtoll** function.

strtod

Converts a number-representing string to a double type floating-point number.

[Format]

```
#include <stdlib.h>
double strtod (const char *nptr, char **endptr);
```

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent a floating-point number

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent a floating-point number: 0
If the string pointed by **nptr** begins with a character that represents a floating-point number: Converted data as a **double** type floating-point number
Abnormal: If the converted data overflows: **HUGE_VAL** with the same sign as that of the string before conversion
If the converted data underflows: 0

[Remarks]

The **strtod** function converts data, from the first digit or the decimal point up to the character immediately before the character that does not represent a floating-point number, into a **double** type floating-point number. However, if neither an exponent nor a decimal point is found in the data to be converted, the compiler assumes that the decimal point comes next to the last digit in the string. In the area pointed by **endptr**, the function sets up a pointer to the first character that does not represent a floating-point number. If some characters that do not represent a floating-point number come before digits, the value of **nptr** is set. If **endptr** is **NULL**, nothing is set in this area.

strtof

Converts a number-representing string to a **float** type floating-point number.

[Format]

#include <stdlib.h>
float strtof (const char *nptr, char **endptr);

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent a floating-point number

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent a floating-point number: 0
If the string pointed by **nptr** begins with a character that represents a floating-point number: Converted data as a **float** type floating-point number
Abnormal: If the converted data overflows: **HUGE_VALF** with the same sign as that of the string before conversion
If the converted data underflows: 0

[Remarks]

If the converted result overflows or underflows, **errno** is set.
The **strtof** function converts data, from the first digit or the decimal point up to the character immediately before the character that does not represent a floating-point number, into a **float** type floating-point number. However, if neither an exponent nor a decimal point is found in the data to be converted, the compiler assumes that the decimal point comes next to the last digit in the string. In the area pointed by **endptr**, the function sets up a pointer to the first character that does not represent a floating-point number. If some characters that do not represent a floating-point number come before digits, the value of **nptr** is set. If **endptr** is **NULL**, nothing is set in this area.

strtold

Converts a number-representing string to a long double type floating-point number.

[Format]

#include <stdlib.h>
long double strtold (const char *nptr, char **endptr);

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent a floating-point number

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent a floating-point number: 0
If the string pointed by **nptr** begins with a character that represents a floating-point number: Converted data as a **long double** type floating-point number
Abnormal: If the converted data overflows: **HUGE_VALL** with the same sign as that of the string before conversion
If the converted data underflows: 0

[Remarks]

If the converted result overflows or underflows, **errno** is set.

The **strtold** function converts data, from the first digit or the decimal point up to the character immediately before the character that does not represent a floating-point number, into a **long double** type floating-point number. However, if neither an exponent nor a decimal point is found in the data to be converted, the compiler assumes that the decimal point comes next to the last digit in the string. In the area pointed by **endptr**, the function sets up a pointer to the first character that does not represent a floating-point number. If some characters that do not represent a floating-point number come before digits, the value of **nptr** is set. If **endptr** is **NULL**, nothing is set in this area.

strtol

Converts a number-representing string to a **long** type integer.

[Format]

#include <stdlib.h>
long strtol (const char *nptr, char **endptr, long base);

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer
base Radix of conversion (0 or 2 to 36)

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0
If the string pointed by **nptr** begins with a character that represents an integer: Converted data as a **long** type integer
Abnormal: If the converted data overflows: **LONG_MAX** or **LONG_MIN** depending on the sign of the string before conversion

[Remarks]

If the converted result overflows, **errno** is set.

The **strtol** function converts data, from the first digit up to the character before the first character that does not represent an integer, into a **long** type integer.

In the storage area pointed by **endptr**, the function sets up a pointer to the first character that does not represent an integer. If some characters that do not represent an integer come before the first digit, the value of **nptr** is set in this area. If **endptr** is **NULL**, nothing is set in this area.

If the value of **base** is 0, the rules described in section 3.1.3 (4), Integers, are observed at conversion. If the value of **base** is 2 to 36, it indicates the radix of conversion, where a (or A) to z (or Z) in the string to be converted correspond to numbers 10 to 35. If a character that is not smaller than the **base** value is found in the string to be converted, conversion stops immediately. A 0 after a sign is ignored at conversion. Similarly, 0x (or 0X) is ignored when **base** is 16.

strtoul

Converts a number-representing string to an unsigned long type integer.

[Format]

#include <stdlib.h>
unsigned long strtoul (const char *nptr, char **endptr, long base);

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer
base Radix of conversion (0 or 2 to 36)

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0
If the string pointed by **nptr** begins with a character that represents an integer: Converted data as an **unsigned long** type integer
Abnormal: If the converted data overflows: **ULONG_MAX**

[Remarks]

If the converted result overflows, **errno** is set.

The **strtoul** function converts data, from the first digit up to the character before the first character that does not represent an integer, into an **unsigned long** type integer.

In the storage area pointed by **endptr**, the function sets up a pointer to the first character that does not represent an integer. If some characters that do not represent an integer come before the first digit, the value of **nptr** is set in this area. If **endptr** is **NULL**, nothing is set in this area.

If the value of **base** is 0, the rules described in section 3.1.3 (4), Integers, are observed at conversion. If the value of **base** is 2 to 36, it indicates the radix of conversion, where a (or A) to z (or Z) in the string to be converted correspond to numbers 10 to 35. If a character that is not smaller than the **base** value is found in the string to be converted, conversion stops immediately. A 0 after a sign is ignored at conversion. Similarly, 0x (or 0X) is ignored when **base** is 16.

strtoll

Converts a number-representing string to a **long long** type integer.

[Format]

#include <stdlib.h>
long long strtoll (const char *nptr, char **endptr, long base);

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer
base Radix of conversion (0 or 2 to 36)

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0
If the string pointed by **nptr** begins with a character that represents an integer: Converted data as a **long long** type integer
Abnormal: If the converted data overflows: **LLONG_MAX** or **LLONG_MIN** depending on the sign of the string before conversion

[Remarks]

If the converted result overflows, **errno** is set.

The **strtoll** function converts data, from the first digit up to the character before the first character that does not represent an integer, into a **long long** type integer.

In the storage area pointed by **endptr**, the function sets up a pointer to the first character that does not represent an integer. If some characters that do not represent an integer come before the first digit, the value of **nptr** is set in this area. If **endptr** is **NULL**, nothing is set in this area.

If the value of **base** is 0, the rules described in section 3.1.3 (4), Integers, are observed at conversion. If the value of **base** is 2 to 36, it indicates the radix of conversion, where a (or A) to z (or Z) in the string to be converted correspond to numbers 10 to 35. If a character that is not smaller than the **base** value is found in the string to be converted, conversion stops immediately. A 0 after a sign is ignored at conversion. Similarly, 0x (or 0X) is ignored when **base** is 16.

strtoull

Converts a number-representing string to an unsigned long long type integer.

[Format]

```
#include <stdlib.h>
unsigned long long strtoull (const char *nptr, char **endptr, long base);
```

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer
base Radix of conversion (0 or 2 to 36)

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0
If the string pointed by **nptr** begins with a character that represents an integer: Converted data as an **unsigned long long** type integer
Abnormal: If the converted data overflows: **ULLONG_MAX**

[Remarks]

If the converted result overflows, **errno** is set.

The **strtoull** function converts data, from the first digit up to the character before the first character that does not represent an integer, into an **unsigned long long** type integer.

In the storage area pointed by **endptr**, the function sets up a pointer to the first character that does not represent an integer. If some characters that do not represent an integer come before the first digit, the value of **nptr** is set in this area. If **endptr** is **NULL**, nothing is set in this area.

If the value of **base** is 0, the rules described in section 3.1.3 (4), Integers, are observed at conversion. If the value of **base** is 2 to 36, it indicates the radix of conversion, where a (or A) to z (or Z) in the string to be converted correspond to numbers 10 to 35. If a character that is not smaller than the **base** value is found in the string to be converted, conversion stops immediately. A 0 after a sign is ignored at conversion. Similarly, 0x (or 0X) is ignored when **base** is 16.

rand

Generates a pseudo-random integer from 0 to **RAND_MAX**.

[Format]

```
#include <stdlib.h>
long rand (void);
```

[Return values]

Pseudo-random integer

srand

Sets an initial value of the pseudo-random number sequence generated by the **rand** function.

[Format]

```
#include <stdlib.h>
void srand (unsigned long seed);
```

[Parameters]

seed Initial value for pseudo-random number sequence generation

[Remarks]

The **srand** function sets up an initial value for pseudo-random number sequence generation of the **rand** function. If pseudo-random number sequence generation by the **rand** function is repeated and if the same initial value is set up again by the **srand** function, the same pseudo-random number sequence is repeated.

If the **rand** function is called before the **srand** function, 1 is set as the initial value for the pseudo-random number generation.

calloc

Allocates a storage area and clears all bits in the allocated storage area to 0.

[Format]

```
#include <stdlib.h>
void *calloc (size_t nelem, size_t elsize);
```

[Parameters]

nelem Number of elements
elsize Number of bytes occupied by a single element

[Return values]

Normal: Starting address of an allocated storage area
Abnormal: Storage allocation failed, or either of the parameter is 0: **NULL**

[Remarks]

The **calloc** function allocates as many storage units of size **elsize** (bytes) as the number specified by nelem. The function also clears all the bits in the allocated storage area to 0.

The CC-RX has a security facility for detecting illegal operations to storage areas. For details, refer to the **-secure_malloc** option in "2.5.4 Library Generator Options".

free

Releases the specified storage area.

[Format]

```
#include <stdlib.h>
void free (void *ptr);
```

[Parameters]

ptr Address of storage area to release

[Remarks]

The **free** function releases the storage area pointed by **ptr**, to enable reallocation for use. If **ptr** is **NULL**, the function carries out nothing.

If the storage area attempted to release was not allocated by the **calloc**, **malloc**, or **realloc** function, or when the area has already been released by the **free** or **realloc** function, correct operation is not guaranteed. Operation result of reference to a released storage area is also not guaranteed.

The CC-RX has a security facility for detecting illegal operations to storage areas. For details, refer to the **-secure_malloc** option in "2.5.4 Library Generator Options".

malloc

Allocates a storage area.

[Format]

```
#include <stdlib.h>
void *malloc (size_t size);
```

[Parameters]

size Size in number of bytes of storage area to allocate

[Return values]

Normal: Starting address of allocated storage area
Abnormal: Storage allocation failed, or **size** is 0: **NULL**

[Remarks]

The **malloc** function allocates a storage area of a specified number of bytes by **size**.
The CC-RX has a security facility for detecting illegal operations to storage areas. For details, refer to the **-secure_malloc** option in "2.5.4 Library Generator Options".

realloc

Changes the size of a storage area to a specified value.

[Format]

```
#include <stdlib.h>
void *realloc (void *ptr, size_t size);
```

[Parameters]

ptr Starting address of storage area to be changed
size Size of storage area in number of bytes after the change

[Return values]

Normal: Starting address of storage area whose size has been changed
Abnormal: Storage area allocation has failed, or size is 0: NULL

[Remarks]

The **realloc** function changes the size of the storage area specified by **ptr** to the number of bytes specified by **size**. If the newly allocated storage area is smaller than the old one, the contents are left unchanged up to the size of the newly allocated area.

When **ptr** is not a pointer to the storage area allocated by the **calloc**, **malloc**, or **realloc** function or when **ptr** is a pointer to the storage area released by the **free** or **realloc** function, operation is not guaranteed.

The CC-RX has a security facility for detecting illegal operations to storage areas. For details, refer to the **-secure_malloc** option in "2.5.4 Library Generator Options".

bsearch

Performs binary search.

[Format]

```
// C
#include <stdlib.h>
void *bsearch (const void *key, const void *base, size_t nmemb, size_t size, long (*compar)(const void *, const void *));

// C++/EC++
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

[Parameters]

key Pointer to data to find
base Pointer to a table to be searched
nmemb Number of members to be searched
size Number of bytes of a member to be searched
compar Pointer to a function that performs comparison

[Return values]

If a matching member is found: Pointer to the matching member
If no matching member is found: **NULL**

[Remarks]

The **bsearch** function searches the table specified by **base** for a member that matches the data specified by **key**, by binary search method. The function that performs comparison should receive pointers **p1** (first parameter) and **p2** (second parameter) to two data items to compare, and return the result complying with the specification below.

*p1 < *p2: Returns a negative value.
*p1 == *p2: Returns 0.
*p1 > *p2: Returns a positive value.

Members to be searched must be placed in the ascending order.

qsort

Performs sorting.

[Format]

```
// C
#include <stdlib.h>
void qsort (const void *base, size_t nmemb, size_t size, long (*compar)(const void *, const void *));

// C++/EC++
#include <stdlib.h>
void qsort(const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

[Parameters]

base Pointer to the table to be sorted
nmemb Number of members to sort
size Number of bytes of a member to be sorted
compar Pointer to a function to perform comparison

[Remarks]

The **qsort** function sorts out data on the table pointed to by **base**. The data arrangement order is specified by the pointer to a function to perform comparison. This comparison function should receive pointers **p1** (first parameter) and **p2** (second parameter) as two data items to be compared, and return the result complying with the specification below.

*p1 < *p2: Returns a negative value.
*p1 == *p2: Returns 0.
*p1 > *p2: Returns a positive value.

abs

Calculates the absolute value of an int type integer.

[Format]

#include <stdlib.h>
long abs (long i);

[Parameters]

i Integer to calculate the absolute value

[Return values]

Absolute value of i

[Remarks]

If the resultant absolute value cannot be expressed as an **int** type integer, correct operation is not guaranteed.

div

Carries out division of **int** type integers and obtains the quotient and remainder.

[Format]

#include <stdlib.h>
div_t div (long numer, long denom);

[Parameters]

numer Dividend
denom Divisor

[Return values]

Quotient and remainder of division of numer by denom.

[Remarks]

Type **div_t** is defined as follows:

```
typedef struct {
    long quot;
    long rem;
} div_t;
```

labs

Calculates the absolute value of a **long** type integer.

[Format]

```
#include <stdlib.h>
long labs (long j);
```

[Parameters]

j Integer to calculate the absolute value

[Return values]

Absolute value of j

[Remarks]

If the resultant absolute value cannot be expressed as a **long** type integer, correct operation is not guaranteed.

ldiv

Carries out division of **long** type integers and obtains the quotient and remainder.

[Format]

#include <stdlib.h>
ldiv_t ldiv (long numer, long denom);

[Parameters]

numer Dividend
denom Divisor

[Return values]

Quotient and remainder of division of **numer** by **denom**.

[Remarks]

Type **ldiv_t** is defined as follows:

```
typedef struct {
    long quot;
    long rem;
} ldiv_t;
```

llabs

Calculates the absolute value of a **long long** type integer.

[Format]

```
#include <stdlib.h>
long long llabs (long long j);
```

[Parameters]

j Integer to calculate the absolute value

[Return values]

Absolute value of j

[Remarks]

If the resultant absolute value cannot be expressed as a **long long** type integer, correct operation is not guaranteed.

lldiv

Carries out division of **long long** type integers and obtains the quotient and remainder.

[Format]

#include <stdlib.h>
lldiv_t lldiv (long long numer, long long denom);

[Parameters]

numer Dividend
denom Divisor

[Return values]

Quotient and remainder of division of **numer** by **denom**.

[Remarks]

Type **lldiv_t** is defined as follows:

```
typedef struct {
    long long quot;
    long long rem;
} lldiv_t;
```

7.4.13 <string.h>

Defines functions for handling character arrays.

Type	Definition Name	Description
Function	memcpy	Copies contents of a source storage area of a specified length to a destination storage area.
	strcpy	Copies contents of a source string including the null character to a destination storage area.
	strncpy	Copies a source string of a specified length to a destination storage area.
	strcat	Concatenates a string after another string.
	strncat	Concatenates a string of a specified length after another string.
	memcmp	Compares two storage areas specified.
	strcmp	Compares two strings specified.
	strncmp	Compares two strings specified for a specified length.
	memchr	Searches a specified storage area for the first occurrence of a specified character.
	strchr	Searches a specified string for the first occurrence of a specified character.
	strcspn	Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are not included in another string specified.
	strpbrk	Searches a specified string for the first occurrence of any character that is included in another string specified.
	strrchr	Searches a specified string for the last occurrence of a specified character.
	strspn	Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are included in another string specified.
	strstr	Searches a specified string for the first occurrence of another string specified.
	strtok	Divides a specified string into some tokens.
	memset	Sets a specified character for a specified number of times at the beginning of a specified storage area.
	strerror	Sets an error message.
	strlen	Calculates the length of a string.
	memmove	Copies contents of a source storage area of a specified length to a destination storage area. Even if a part of the source storage area and a part of the destination storage area overlap, correct copy is performed.

Implementation-Defined Specifications

Item	Compiler Specifications
Error message returned by the **strerror** function	Refer to section 10.5.6, Standard Library Error Messages.

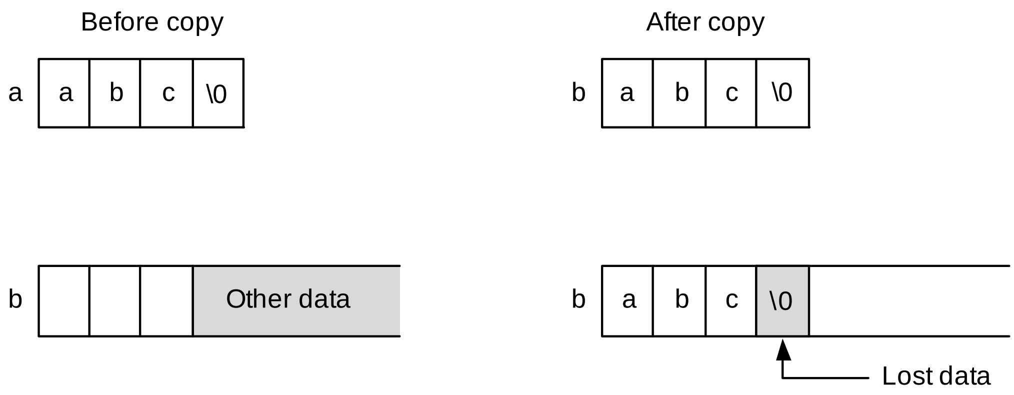
When using functions defined in this standard include file, note the following.

(1) On copying a string, if the destination area is smaller than the source area, correct operation is not guaranteed.

Example

```
char a[]="abc";
char b[3];
      .
      .
      .
strcpy (b, a);
```

In the above example, the size of array **a** (including the null character) is 4 bytes. Copying by **strcpy** overwrites data beyond the boundary of array **b**.

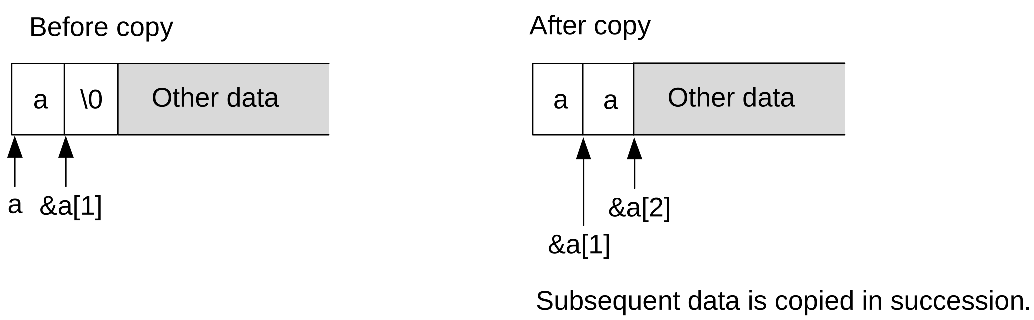


(2) On copying a string, if the source area overlaps the destination area, correct operation is not guaranteed.

Example

```
int a[ ]="a";
    :
    :
strcpy(&a[1], a);
    :
```

In the above example, before the null character of the source is read, 'a' is written over the null character. Then the subsequent data after the source string is overwritten in succession.



memcpy

Copies the contents of a source storage area of a specified length to a destination storage area.

[Format]

```
#include <string.h>
void *memcpy (void *s1, const void *s2, size_t n);
```

[Parameters]

s1 Pointer to destination storage area
s2 Pointer to source storage area
n Number of characters to be copied

[Return values]

s1 value

strcpy

Copies the contents of a source string including the null character to a destination storage area.

[Format]

```
#include <string.h>
char *strcpy (char *s1, const char *s2);
```

[Parameters]

s1 Pointer to destination storage area
s2 Pointer to source string

[Return values]

s1 value

strncpy

Copies a source string of a specified length to a destination storage area.

[Format]

```
#include <string.h>
char *strncpy (char *s1, const char *s2, size_t n);
```

[Parameters]

s1 Pointer to destination storage area
s2 Pointer to source string
n Number of characters to be copied

[Return values]

s1 value

[Remarks]

The **strncpy** function copies up to **n** characters from the beginning of the string pointed by **s2** to a storage area pointed by **s1**. If the length of the string specified by **s2** is shorter than **n** characters, the function elongates the string to the length by padding with null characters.

If the length of the string specified by **s2** is longer than **n** characters, the copied string in **s1** storage area ends with a character other than the null character.

strcat

Concatenates a string after another string.

[Format]

```
#include <string.h>
char *strcat (char *s1, const char *s2);
```

[Parameters]

s1 Pointer to the string after which another string is appended
s2 Pointer to the string to be appended after the other string

[Return values]

s1 value

[Remarks]

The **strcat** function concatenates the string specified by **s2** at the end of another string specified by **s1**. The null character indicating the end of the **s2** string is also copied. The null character at the end of the **s1** string is deleted.

strncat

Concatenates a string of a specified length after another string.

[Format]

```
#include <string.h>
char *strncat (char *s1, const char *s2, size_t n);
```

[Parameters]

s1 Pointer to the string after which another string is appended
s2 Pointer to the string to be appended after the other string
n Number of characters to concatenate

[Return values]

s1 value

[Remarks]

The **strncat** function concatenates up to **n** characters from the beginning of the string specified by **s2** at the end of another string specified by **s1**. The null character at the end of the **s1** string is replaced by the first character of the **s2** string. A null character is appended to the end of the concatenated string.

memcmp

Compares the contents of two storage areas specified.

[Format]

```
#include <string.h>
long memcmp (const void *s1, const void *s2, size_t n);
```

[Parameters]

s1 Pointer to the reference storage area to be compared
s2 Pointer to the storage area to compare to the reference
n Number of characters to compare

[Return values]

If storage area pointed by **s1** > storage area pointed by **s2**: Positive value
If storage area pointed by **s1** == storage area pointed by **s2**: 0
If storage area pointed by **s1** < storage area pointed by **s2**: Negative value

[Remarks]

The **memcmp** function compares the contents of the first **n** characters in the storage areas pointed by **s1** and **s2**. The rules of comparison are implementation-defined.

strcmp

Compares the contents of two strings specified.

[Format]

```
#include <string.h>
long strcmp (const char *s1, const char *s2);
```

[Return values]

If string pointed by **s1** > string pointed by **s2**: Positive value
If string pointed by **s1** == string pointed by **s2**: 0
If string pointed by **s1** < string pointed by **s2**: Negative value

[Parameters]

s1 Pointer to the reference string to be compared
s2 Pointer to the string to compare to the reference

[Remarks]

The **strcmp** function compares the contents of the strings pointed by **s1** and **s2**, and sets up the comparison result as a return value. The rules of comparison are implementation-defined.

strncmp

Compares two strings specified up to a specified length.

[Format]

```
#include <string.h>
long strncmp (const char *s1, const char *s2, size_t n);
```

[Parameters]

s1 Pointer to the reference string to be compared
s2 Pointer to the string to compare to the reference
n Maximum number of characters to compare

[Return values]

If string pointed by **s1** > string pointed by **s2**: Positive value
If string pointed by **s1** == string pointed by **s2**: 0
If string pointed by **s1** < string pointed by **s2**: Negative value

[Remarks]

The **strncmp** function compares the contents of the strings pointed by **s1** and **s2**, up to **n** characters. The rules of comparison are implementation-defined.

memchr

Searches a specified storage area for the first occurrence of a specified character.

[Format]

```
#include <string.h>
void *memchr (const void *s, long c, size_t n);
```

[Parameters]

s Pointer to the storage area to be searched
c Character to search for
n Number of characters to search

[Return values]

If the character is found: Pointer to the found character
If the character is not found: **NULL**

[Remarks]

The **memchr** function searches the storage area specified by **s** from the beginning up to **n** characters, looking for the first occurrence of the character specified as **c**. If the **c** character is found, the function returns the pointer to the found character.

strchr

Searches a specified string for the first occurrence of a specified character.

[Format]

char *strchr (const char *s, long c);

[Parameters]

s Pointer to the string to be searched
c Character to search for

[Return values]

If the character is found: Pointer to the found character
If the character is not found: **NULL**

[Remarks]

The **strchr** function searches the string specified by **s** looking for the first occurrence of the character specified as **c**. If the **c** character is found, the function returns the pointer to the found character.

The null character at the end of the **s** string is included in the search object.

strcspn

Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are not included in another string specified.

[Format]

```
#include <string.h>
size_t strcspn (const char *s1, const char *s2);
```

[Parameters]

s1 Pointer to the string to be checked
s2 Pointer to the string used to check **s1**

[Return values]

Number of characters at the beginning of the **s1** string that are not included in the **s2** string

[Remarks]

The **strcspn** function checks from the beginning of the string specified by **s1**, counts the number of consecutive characters that are not included in another string specified by **s2**, and returns that length.

The null character at the end of the **s2** string is not taken as a part of the **s2** string.

strpbrk

Searches a specified string for the first occurrence of the character that is included in another string specified.

[Format]

```
#include <string.h>
char *strpbrk (const char *s1, const char *s2);
```

[Parameters]

s1 Pointer to the string to be searched
s2 Pointer to the string that indicates the characters to search **s1** for

[Return values]

If the character is found: Pointer to the found character
If the character is not found: **NULL**

[Remarks]

The **strpbrk** function searches the string specified by **s1** looking for the first occurrence of any character included in the string specified by **s2**. If any searched character is found, the function returns the pointer to the first occurrence.

strrchr

Searches a specified string for the last occurrence of a specified character.

[Format]

```
#include <string.h>
char *strrchr (const char *s, long c);
```

[Parameters]

s Pointer to the string to be searched
c Character to search for

[Return values]

If the character is found: Pointer to the found character
If the character is not found: **NULL**

[Remarks]

The **strrchr** function searches the string specified by **s** looking for the last occurrence of the character specified by **c**. If the **c** character is found, the function returns the pointer to the last occurrence of that character.

The null character at the end of the **s** string is included in the search objective.

strspn

Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are included in another string specified.

[Format]

```
#include <string.h>
size_t strspn (const char *s1, const char *s2);
```

[Parameters]

s1 Pointer to the string to be checked
s2 Pointer to the string used to check **s1**

[Return values]

Number of characters at the beginning of the **s1** string that are included in the **s2** string

[Remarks]

The **strspn** function checks from the beginning of the string specified by **s1**, counts the number of consecutive characters that are included in another string specified by **s2**, and returns that length.

strstr

Searches a specified string for the first occurrence of another string specified.

[Format]

```
#include <string.h>
char *strstr (const char *s1, const char *s2);
```

[Parameters]

s1 Pointer to the string to be searched
s2 Pointer to the string to search for

[Return values]

If the string is found: Pointer to the found string
If the string is not found: **NULL**

[Remarks]

The **strstr** function searches the string specified by **s1** looking for the first occurrence of another string specified by **s2**, and returns the pointer to the first occurrence.

strtok

Divides a specified string into some tokens.

[Format]

#include <string.h>
char *strtok (char *s1, const char *s2);

[Return values]

If division into tokens is successful: Pointer to the first token divided
If division into tokens is unsuccessful: **NULL**

[Parameters]

s1 Pointer to the string to be divided into some tokens
s2 Pointer to the string representing string-dividing characters

[Remarks]

The **strtok** function should be repeatedly called to divide a string.

(a) First call
The string pointed by **s1** is divided at a character included in the string pointed by **s2**. If a token has been separated, the function returns a pointer to the beginning of that token. Otherwise, the function returns **NULL**.

(b) Second and subsequent calls
Starting from the next character separated before as the token, the function repeats division at a character included in the string pointed by **s2**. If a token has been separated, the function returns a pointer to the beginning of that token. Otherwise, the function returns **NULL**.
At the second and subsequent calls, specify **NULL** as the first parameter. The string pointed by **s2** can be changed at each call. The null character is appended at the end of a separated token.
An example of use of the **strtok** function is shown below.

Example

```
1  #include <string.h>
2  static char s1[ ]="a@b, @c/@d";
3  char *ret;
4
5  ret = strtok(s1, "@");
6  ret = strtok(NULL, ",@");
7  ret = strtok(NULL, "/@");
8  ret = strtok(NULL, "@");
```

Explanation:
The above example program uses the **strtok** function to divide string "a@b, @c/@d" into tokens a, b, c, and d.
The second line specifies string "a@b, @c/@d" as an initial value for string **s1**.
The fifth line calls the **strtok** function to divide tokens using '@' as the delimiter. As a result, a pointer to character 'a' is returned, and the null character is embedded at '@,' the first delimiter after character 'a.' Thus string 'a' has been separated.
Specify **NULL** for the first parameter to consecutively separate tokens from the same string, and repeat calling the **strtok** function.
Consequently, the function separates strings 'b,' 'c,' and 'd.'

memset

Sets a specified character a specified number of times at the beginning of a specified storage area.

[Format]

```
#include <string.h>
void *memset (void *s, long c,  size_t n);
```

[Parameters]

s Pointer to storage area to set characters in
c Character to be set
n Number of characters to be set

[Return values]

Value of s

[Remarks]

The **memset** function sets the character specified by **c** a number of times specified by **n** in the storage area specified by **s**.

strerror

Returns an error message corresponding to a specified error number.

[Format]

```
#include <string.h>
char *strerror (long s);
```

[Parameters]

s Error number

[Return values]

Pointer to the error message (string) corresponding to the specified error number

[Remarks]

The strerror function receives an error number specified by **s** and returns an error message corresponding to the number. Contents of error messages are implementation-defined.

If the returned error message is modified, correct operation is not guaranteed.

strlen

Calculates the length of a string.

[Format]

```
#include <string.h>
size_t strlen (const char *s);
```

[Parameters]

s Pointer to the string to check the length of

[Return values]

Number of characters in the string

[Remarks]

The null character at the end of the **s** string is excluded from the string length.

memmove

Copies the specified size of the contents of a source area to a destination storage area. If part of the source storage area and the destination storage area overlap, data is copied to the destination storage area before the overlapped source storage area is overwritten. Therefore, correct copy is enabled.

[Format]

```
#include <string.h>
void *memmove (void *s1, const void *s2, size_t n);
```

[Parameters]

s1 Pointer to the destination storage area
s2 Pointer to the source storage area
n Number of characters to be copied

[Return values]

Value of **s1**

7.4.14 <complex.h>

Performs various complex number operations. For **double**-type complex number functions, the definition names are used as function names without change. For **float**-type and **long double**-type function names, "f" and "l" are added to the end of definition names, respectively.

Type	Definition Name	Description
Function	cacosf / cacos / cacosl *<-lang=c99>*	Calculates the arc cosine of a complex number.
	casinf / casin / casinl *<-lang=c99>*	Calculates the arc sine of a complex number.
	catanf / catan / catanl *<-lang=c99>*	Calculates the arc tangent of a complex number.
	ccosf / ccos / ccosl *<-lang=c99>*	Calculates the cosine of a complex number.
	csinf / csin / csinl *<-lang=c99>*	Calculates the sine of a complex number.
	ctanf / ctan / ctanl *<-lang=c99>*	Calculates the tangent of a complex number.
	cacoshf / cacosh / cacoshl *<-lang=c99>*	Calculates the arc hyperbolic cosine of a complex number.
	casinhf / casinh / casinhl *<-lang=c99>*	Calculates the arc hyperbolic sine of a complex number.
	catanhf / catanh / catanhl *<-lang=c99>*	Calculates the arc hyperbolic tangent of a complex number.
	ccoshf / ccosh / ccoshl *<-lang=c99>*	Calculates the hyperbolic cosine of a complex number.
	csinhf / csinh / csinhl *<-lang=c99>*	Calculates the hyperbolic sine of a complex number.
	ctanhf / ctanh / ctanhl *<-lang=c99>*	Calculates the hyperbolic tangent of a complex number.
	cexpf / cexp / cexpl *<-lang=c99>*	Calculates the natural logarithm base **e** raised to the complex power 2.
	clogf / clog / clogl *<-lang=c99>*	Calculates the natural logarithm of a complex number.
	cabsf / cabs / cabsl *<-lang=c99>*	Calculates the absolute value of a complex number.
	cpowf / cpow / cpowl *<-lang=c99>*	Calculates a power of a complex number.
	csqrtf / csqrt / csqrtl *<-lang=c99>*	Calculates the square root of a complex number.
	cargf / carg / cargl *<-lang=c99>*	Calculates the argument of a complex number.
	cimagf / cimag / cimagl *<-lang=c99>*	Calculates the imaginary part of a complex number.
	conjf / conj / conjl *<-lang=c99>*	Reverses the sign of the imaginary part and calculates the complex conjugate of a complex number.
	cprojf / cproj / cprojl *<-lang=c99>*	Calculates the projection of a complex number on Riemann sphere.
	crealf / creal / creall *<-lang=c99>*	Calculates the real part of a complex number.

cacosf / cacos / cacosl

Calculates the arc cosine of a complex number.

[Format]

```
#include <complex.h>
float complex cacosf(float complex z);
double complex cacos(double complex z);
long double complex cacosl(long double complex z);
```

[Parameters]

z Complex number for which arc cosine is to be computed

[Return values]

Complex arc cosine of **z**

[Remarks]

The **cacos** function returns the arc cosine in the range $[0, \pi]$ on the real axis and in the infinite range on the imaginary axis.

casinf / casin / casinl

Calculates the arc sine of a complex number.

[Format]

```
#include <complex.h>
float complex casinf(float complex z);
double complex casin(double complex z);
long double complex casinl(long double complex z);
```

[Parameters]

z Complex number for which arc sine is to be computed

[Return values]

Complex arc sine of **z**

[Remarks]

The **casin** function returns the arc sine in the range [–π/2, π/2] on the real axis and in the infinite range on the imaginary axis.

catanf / catan / catanl

Calculates the arc tangent of a complex number.

[Format]

```
#include <complex.h>
float complex catanf(float complex z);
double complex catan(double complex z);
long double complex catanl(long double complex z);
```

[Parameters]

z Complex number for which arc tangent is to be computed

[Return values]

Complex arc tangent of **z**

[Remarks]

The **catan** function returns the arc tangent in the range [–π/2, π/2] on the real axis and in the infinite range on the imaginary axis.

ccosf / ccos / ccosl

Calculates the cosine of a complex number.

[Format]

```
#include <complex.h>
float complex ccosf(float complex z);
double complex ccos(double complex z);
long double complex ccosl(long double complex z);
```

[Parameters]

z Complex number for which cosine is to be computed

[Return values]

Complex cosine of **z**

csinf / csin / csinl

Calculates the sine of a complex number.

[Format]

```
#include <complex.h>
float complex csinf(float complex z);
double complex csin(double complex z);
long double complex csinl(long double complex z);
```

[Parameters]

z Complex number for which sine is to be computed

[Return values]

Complex sine of **z**

ctanf / ctan / ctanl

Calculates the tangent of a complex number.

[Format]

```
#include <complex.h>
float complex ctanf(float complex z);
double complex ctan(double complex z);
long double complex ctanl(long double complex z);
```

[Parameters]

z Complex number for which tangent is to be computed

[Return values]

Complex tangent of **z**

cacoshf / cacosh / cacoshl

Calculates the arc hyperbolic cosine of a complex number.

[Format]

```
#include <complex.h>
float complex cacoshf(float complex z);
double complex cacosh(double complex z);
long double complex cacoshl(long double complex z);
```

[Parameters]

z Complex number for which arc hyperbolic cosine is to be computed

[Return values]

Complex arc hyperbolic cosine of **z**

[Remarks]

The **cacoshf** function returns the arc hyperbolic cosine in the range [0, π].

casinhf / casinh / casinhl

Calculates the arc hyperbolic sine of a complex number.

[Format]

```
#include <complex.h>
float complex casinhf(float complex z);
double complex casinh(double complex z);
long double complex casinhl(long double complex z);
```

[Parameters]

z Complex number for which arc hyperbolic sine is to be computed

[Return values]

Complex arc hyperbolic sine of **z**

catanhf / catanh / catanhl

Calculates the arc hyperbolic tangent of a complex number.

[Format]

```
#include <complex.h>
float complex catanhf(float complex z);
double complex catanh(double complex z);
long double complex catanhl(long double complex z);
```

[Parameters]

z Complex number for which arc hyperbolic tangent is to be computed

[Return values]

Complex arc hyperbolic tangent of **z**

ccoshf / ccosh / ccoshl

Calculates the hyperbolic cosine of a complex number.

[Format]

```
#include <complex.h>
float complex ccoshf(float complex z);
double complex ccosh(double complex z);
long double complex ccoshl(long double complex z);
```

[Parameters]

z Complex number for which hyperbolic cosine is to be computed

[Return values]

Complex hyperbolic cosine of **z**

csinhf / csinh / csinhl

Calculates the hyperbolic sine of a complex number.

[Format]

```
#include <complex.h>
float complex csinhf(float complex z);
double complex csinh(double complex z);
long double complex csinhl(long double complex z);
```

[Parameters]

z Complex number for which hyperbolic sine is to be computed

[Return values]

Complex hyperbolic sine of **z**

ctanhf / ctanh / ctanhl

Calculates the hyperbolic tangent of a complex number.

[Format]

```
#include <complex.h>
float complex ctanhf(float complex z);
double complex ctanh(double complex z);
long double complex ctanhl(long double complex z);
```

[Parameters]

z Complex number for which hyperbolic tangent is to be computed

[Return values]

Complex hyperbolic tangent of **z**

cexpf / cexp / cexpl

Calculates the exponential function value of a complex number.

[Format]

```
#include <complex.h>
float complex cexpf(float complex z);
double complex cexp(double complex z);
long double complex cexpl(long double complex z);
```

[Parameters]

z Complex number for which exponential function is to be computed

[Return values]

Exponential function value of **z**

clogf / clog / clogl

Calculates the natural logarithm of a complex number.

[Format]

#include <complex.h>
float complex clogf(float complex z);
double complex clog(double complex z);
long double complex clogl(long double complex z);

[Parameters]

z Complex number for which natural logarithm is to be computed

[Return values]

Normal: Natural logarithm of **z**
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if z is negative.
A range error occurs if z is 0.0.
The **clog** function returns the natural logarithm in the infinite range on the real axis and in the range [–iπ, +iπ] on the imaginary axis.

cabsf / cabs / cabsl

Calculates the absolute value of a complex number.

[Format]

```
#include <complex.h>
float cabsf(float complex z);
double cabs(double complex z);
long double cabsl(long double complex z);
```

[Parameters]

z Complex number for which absolute value is to be computed

[Return values]

Absolute value of **z**

cpowf / cpow / cpowl

Calculates a power of a complex number.

[Format]

```
#include <complex.h>
float complex cpowf(float complex x, float complex y);
double complex cpow(double complex x, double complex y);
long double complex cpowl(long double complex x, long double complex y);
```

[Parameters]

x Value to be raised to a power
y Power value

[Return values]

Normal: Value of **x** raised to the power **y**
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if **x** is 0.0 and **y** is 0.0 or smaller, or if **x** is negative and **y** is not an integer.
The branch cut for the first parameter of the **cpow** function group is along the negative real axis.

csqrtf / csqrt / csqrtl

Calculates the square root of a complex number.

[Format]

```
#include <complex.h>
float complex csqrtf(float complex z);
double complex csqrt(double complex z);
long double complex csqrtl(long double complex z);
```

[Parameters]

z Complex number for which the square root is to be computed

[Return values]

Normal: Complex square root of **z**
Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs if **z** is negative.
The branch cut for the **csqrt** function group is along the negative real axis.
The range of the return value from the **csqrt** function group is the right halfplane including the imaginary axis.

cargf / carg / cargl

Calculates the argument.

[Format]

```
#include <complex.h>
float cargf(float complex z);
double carg(double complex z);
long double cargl(long double complex z);
```

[Parameters]

z Complex number for which the argument is to be computed

[Return values]

Argument value of **z**

[Remarks]

The branch cut for the **carg** function group is along the negative real axis.
The **carg** function group returns the argument in the range $[-\pi, +\pi]$.

cimagf / cimag / cimagl

Calculates the imaginary part.

[Format]

```
#include <complex.h>
float cimagf(float complex z);
double cimag(double complex z);
long double cimagl(long double complex z);
```

[Parameters]

z Complex number for which the imaginary part is to be computed

[Return values]

Imaginary part value of **z** as a real number

conjf / conj / conjl

Reverses the sign of the imaginary part of a complex number and calculates the complex conjugate.

[Format]

```
#include <complex.h>
float complex conjf(float complex z);
double complex conj(double complex z);
long double complex conjl(long double complex z);
```

[Parameters]

z Complex number for which the complex conjugate is to be computed

[Return values]

Complex conjugate of **z**

cprojf / cproj / cprojl

Calculates the projection of a complex number on the Riemann sphere.

[Format]

```
#include <complex.h>
float complex cprojf(float complex z);
double complex cproj(double complex z);
long double complex cprojl(long double complex z);
```

[Parameters]

z Complex number for which the projection on the Riemann sphere is to be computed

[Return values]

Projection of **z** on the Riemann sphere

crealf / creal / creall

Calculates the real part of a complex number.

[Format]

```
#include <complex.h>
float crealf(float complex z);
double creal(double complex z);
long double creall(long double complex z);
```

[Parameters]

z Complex number for which the real part value is to be computed

[Return values]

Real part value of **z**

7.4.15 <fenv.h>

Provides access to the floating-point environment.
The following macros and functions are all implementation-defined.

Type	Definition Name	Description
Type (macro)	fenv_t	Indicates the type of the entire floating-point environment.
	fexcept_t	Indicates the type of the floating-point status flags.
Constant (macro)	FE_DIVBYZERO FE_INEXACT FE_INVALID FE_OVERFLOW FE_UNDERFLOW FE_ALL_EXCEPT	Indicates the values (macros) defined when the floating-point exception is supported.
	FE_DOWNWARD FE_TONEAREST FE_TOWARDZERO FE_UPWARD	Indicates the values (macros) of the floating-point rounding direction.
	FE_DFL_ENV	Indicates the default floating-point environment of the program.
Function	feclearexcept	Attempts to clear a floating-point exception.
	fegetexceptflag	Attempts to store the state of a floating-point flag in an object.
	feraiseexcept	Attempts to generate a floating-point exception.
	fesetexceptflag	Attempts to set a floating-point flag.
	fetestexcept	Checks if floating-point flags are set.
	fegetround	Gets the rounding direction.
	fesetround	Sets the rounding direction.
	fegetenv	Attempts to get the floating-point environment.
	feholdexcept	Saves the floating-point environment, clears the floating-point status flags, and sets the non-stop mode for the floating-point exceptions.
	fesetenv	Attempts to set the floating-point environment.
	feupdateenv	Attempts to save the floating-point exceptions in the automatic storage, set the floating-point environment, and generate the saved floating-point exceptions.

feclearexcept

Attempts to clear a floating-point exception.

[Format]

```
#include <fenv.h>
long feclearexcept(long e);
```

[Parameters]

e Floating-point exception

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fegetexceptflag

Gets the state of a floating-point flag.

[Format]

```
#include <fenv.h>
long fegetexceptflag(fexcept_t *f, long e);
```

[Parameters]

f Pointer to area to store the exception flag state
e Value indicating the exception flag whose state is to be acquired

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

feraiseexcept

Attempts to generate a floating-point exception.

[Format]

```
#include <fenv.h>
long feraiseexcept(long e);
```

[Parameters]

e Value indicating the exception to be generated

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

When generating an "overflow" or "underflow" floating-point exception, whether the **feraiseexcept** function also generates an "inexact" floating-point exception is implementation-defined.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fesetexceptflag

Sets the state of an exception flag.

[Format]

#include <fenv.h>
long fesetexceptflag(const fexcept_t *f, long e);

[Parameters]

f Pointer to the source location from which the exception flag state is to be acquired
e Value indicating the exception flag whose state is to be set

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

Before calling the **fesetexceptflag** function, specify a flag state in the source location through the **fegetexceptflag** function.

The **fesetexceptflag** function only sets the flag state without generating the corresponding floating-point exception.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fetestexcept

Checks the exception flag states.

[Format]

```
#include <fenv.h>
long fetestexcept(long e);
```

[Parameters]

e Value indicating flags whose states are to be checked (multiple flags can be specified)

[Return values]

Bitwise OR of **e** and floating-point exception macros

[Remarks]

A single **fetestexcept** function call can check multiple floating-point exceptions.
Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fegetround

Gets the current rounding direction.

[Format]

```
#include <fenv.h>
long fegetround(void);
```

[Return values]

Normal: 0
Abnormal: Negative value when there is no rounding direction macro value or the rounding direction cannot be determined

[Remarks]

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fesetround

Sets the current rounding direction.

[Format]

```
#include <fenv.h>
#include <assert.h>
long fesetround(long rnd);
```

[Return values]

0 only when the rounding direction has been set successfully

[Remarks]

The rounding direction is not changed if the rounding direction requests through the **fesetround** function differs from the rounding macro value.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fegetenv

Gets the floating-point environment.

[Format]

```
#include <fenv.h>
long fegetenv( fenv_t *f);
```

[Parameters]

f Pointer to area to store the floating-point environment

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

feholdexcept

Saves the floating-point environment.

[Format]

```
#include <fenv.h>
long feholdexcept(fenv_t *f);
```

[Parameters]

f Pointer to the floating-point environment

[Return values]

0 only when the environment has been saved successfully

[Remarks]

When saving the floating-point function environment, the **feholdexcept** function clears the floating-point status flags and sets the non-stop mode for all floating-point exceptions. In non-stop mode, execution continues even after a floating-point exception occurs.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fesetenv

Sets the floating-point environment.

[Format]

```
#include <fenv.h>
long fesetenv(const fenv_t *f);
```

[Parameters]

f Pointer to the floating-point environment

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

For the argument of this function, specify the environment stored or saved by the **fegetenv** or **feholdexcept** function, or the environment equal to the floating-point environment macro.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

feupdateenv

Sets the floating-point environment with the previously generated exceptions retained.

[Format]

```
#include <fenv.h>
long feupdateenv(const fenv_t *f);
```

[Parameters]

f Pointer to the floating-point environment to be set

[Return values]

Normal: 0
Abnormal: Nonzero

[Remarks]

For the argument of this function, specify the object stored or saved by the **fegetenv** or **feholdexcept** function call, or the floating-point environment equal to the floating-point environment macro.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

7.4.16 <inttypes.h>

Extends the integer types.
The following macros and functions are all implementation-defined.

Type	Definition Name	Description
Type (macro)	imaxdiv_t	Indicates the type of the value returned by the **imaxdiv** function.

Type	Definition Name	Description
Variable (macro)	PRIdN PRIdLEASTN PRIdFASTN PRIdMAX PRIdPTR PRIiN PRIiLEASTN PRIiFASTN PRIiMAX PRIiPTR PRIoN PRIoLEASTN PRIoFASTN PRIoMAX PRIoPTR PRIuN PRIuLEASTN PRIuFASTN PRIuMAX PRIuPTR PRIxN PRIxLEASTN PRIxFASTN PRIxMAX PRIxPTR PRIXN PRIXLEASTN PRIXFASTN PRIXMAX PRIXPTR SCNdN SCNdLEASTN SCNdFASTN SCNdMAX SCNdPTR SCNiN SCNiLEASTN SCNiFASTN SCNiMAX SCNiPTR SCNoN SCNoLEASTN SCNoFASTN SCNoMAX SCNoPTR SCNuN SCNuLEASTN SCNuFASTN SCNuMAX SCNuPTR SCNxN SCNxLEASTN SCNxFASTN SCNxMAX SCNxPTR	

Type	Definition Name	Description
Function	imaxabs	Calculates the absolute value.
	imaxdiv	Calculates the quotient and remainder.
	strtoimax / strtoumax	Equivalent to the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions, except that the initial part of the string is converted to **intmax_t** and **uintmax_t** representation.
	wcstoimax / wcstoumax	Equivalent to the **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions except that the initial part of the wide string is converted to **intmax_t** and **uintmax_t** representation.

imaxabs

Calculates the absolute value.

[Format]

```
#include <inttypes.h>
intmax_t imaxabs(intmax_t a);
```

[Parameters]

a Value for which the absolute value is to be computed

[Return values]

Absolute value of **a**

imaxdiv

Performs "/" and "%" operations for the same operands simultaneously.

[Format]

#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t n, intmax_t d);

[Parameters]

n The left hand side operand of "/" or "%" operator
d The right hand side operand of "/" or "%" operator

[Return values]

An object typed as struct imaxdiv_t, which has members quot (the quotient) and rem (the remainder).

[Remarks]

Type **imaxdiv_t** is defined as follows:

```
typedef struct {
    long long quot;
    long long rem;
} imaxdiv_t;
```

strtoimax / strtoumax

Converts a number-representing string to an **intmax_t type** integer.

[Format]

#include <inttypes.h>
intmax_t strtoimax(const char *nptr, char **endptr, long base);
uintmax_t strtoumax(const char *nptr, char **endptr, long base);

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer
base Radix of conversion (0 or 2 to 36)

[Return Values]

Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0
If the string pointed by **nptr** begins with a character that represents an integer: Converted data as an **intmax_t** type integer
Abnormal: If the converted data overflows: **INTMAX_MAX**, **INTMAX_MIN**, or **UINTMAX_MAX**

[Remarks]

If the converted result overflows, **ERANGE** is set in **errno**.

The **strtoimax** and **strtoumax** functions are equivalent to the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions except that the initial part of the string is respectively converted to **intmax_t** and **uintmax_t** integers.

wcstoimax / wcstoumax

Converts a number-representing string to an **intmax_t** or **uintmax_t** type integer.

[Format]

```
#include <stddef.h>
#include <inttypes.h>
intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);
uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);
```

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer
base Radix of conversion (0 or 2 to 36)

[Return Values]

Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0
If the string pointed by **nptr** begins with a character that represents an integer: Converted data as an **intmax_t** type integer
Abnormal: If the converted data overflows: **INTMAX_MAX**, **INTMAX_MIN**, or **UINTMAX_MAX**

[Remarks]

If the converted result overflows, **ERANGE** is set in **errno**.

The **wcstoimax** and **wcstoumax** functions are equivalent to the **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions, except that the initial part of the string is respectively converted to **intmax_t** and **uintmax_t** integers.

7.4.17 <iso646.h>

This header file defines macros only.

<table>
<tr><th>Type</th><th>Definition Name</th><th>Description</th></tr>
<tr><td rowspan="11">Macro</td><td>and</td><td>&&</td></tr>
<tr><td>and_eq</td><td>&=</td></tr>
<tr><td>bitand</td><td>&</td></tr>
<tr><td>bitor</td><td>|</td></tr>
<tr><td>compl</td><td>~</td></tr>
<tr><td>not</td><td>!</td></tr>
<tr><td>not_eq</td><td>!=</td></tr>
<tr><td>or</td><td>||</td></tr>
<tr><td>or_eq</td><td>|=</td></tr>
<tr><td>xor</td><td>^</td></tr>
<tr><td>xor_eq</td><td>^=</td></tr>
</table>

7.4.18 <stdbool.h>

This header file defines macros only.

Type	Definition Name	Description
Macro (variable)	bool	Expanded to _**Bool**.
Macro (constant)	true	Expanded to 1.
	false	Expanded to 0.
	__bool_true_false_are_defined	Expanded to 1.

7.4.19 <stdint.h>

This header file defines macros only.

Type	Definition Name	Description
Macro	int_least8_t uint_least8_t int_least16_t uint_least16_t int_least32_t uint_least32_t int_least64_t uint_least64_t	Indicates the types whose size is large enough to store signed and unsigned integer types of 8, 16, 32, and 64 bits.
	int_fast8_t uint_fast8_t int_fast16_t uint_fast16_t int_fast32_t uint_fast32_t int_fast64_t uint_fast64_t	Indicates the types which can operate signed and unsigned integer types of 8, 16, 32, and 64 bits at the fastest speed.
	intptr_t uintptr_t	These indicate signed and unsigned integer types that can be converted to or from pointers to **void**.
	intmax_t uintmax_t	These indicate signed and unsigned integer types that can represent all signed and unsigned integer types.
	intN_t uintN_t	These indicate **N**-bit signed and unsigned inter types.
	INT*N*_MIN INT*N*_MAX UINT*N*_MAX	Indicates the minimum value of exact-width signed integer type. Indicates the maximum value of exact-width signed integer type. Indicates the maximum value of exact-width unsigned integer type.
	INT_LEAST*N*_MIN INT_LEAST*N*_MAX UINT_LEAST*N*_MAX	Indicates the minimum value of minimum-width signed integer type. Indicates the maximum value of minimum-width signed integer type. Indicates the maximum value of minimum-width unsigned integer type.
	INT_FAST*N*_MIN INT_FAST*N*_MAX UINT_FAST*N*_MAX	Indicates the minimum value of fastest minimum-width signed integer type. Indicates the maximum value of fastest minimum-width signed integer type. Indicates the maximum value of fastest minimum-width unsigned integer type.
	INTPTR_MIN INTPTR_MAX UINTPTR_MAX	Indicates the minimum value of pointer-holding signed integer type. Indicates the maximum value of pointer-holding signed integer type. Indicates the maximum value of pointer-holding unsigned integer type.
	INTMAX_MIN INTMAX_MAX UINTMAX_MAX	Indicates the minimum value of greatest-width signed integer type. Indicates the maximum value of greatest-width signed integer type. Indicates the maximum value of greatest-width unsigned integer type.
	PTRDIFF_MIN PTRDIFF_MAX	-2147483648 +2147483647
	SIG_ATOMIC_MIN SIG_ATOMIC_MAX	-2147483648 +2147483647
	SIZE_MAX	4294967295

Type	Definition Name	Description
Macro	WCHAR_MIN WCHAR_MAX	0 65535U
	WINT_MIN WINT_MAX	0 65535U
Function (macro)	INT*N*_C UINT*N*_C	Expanded to an integer constant expression corresponding to **Int_leastN_t**. Expanded to an integer constant expression corresponding to **Uint_leastN_t**.
	INT_MAX_C UINT_MAX_C	Expanded to an integer constant expression with type **intmax_t**. Expanded to an integer constant expression with type **uintmax_t**.

7.4.20 <tgmath.h>

This header file defines macros only.

Type-Generic Macro	<math.h> Functions	<complex.h> Functions
acos	acos	cacos
asin	asin	casin
atan	atan	catan
acosh	acosh	cacosh
asinh	asinh	casinh
atanh	atanh	catanh
cos	cos	ccos
sin	sin	csin
tan	tan	ctan
cosh	cosh	ccosh
sinh	sinh	csinh
tanh	tanh	ctanh
exp	exp	cexp
log	log	clog
pow	pow	cpow
sqrt	sqrt	csqrt
fabs	fabs	cfabs
atan2	atan2	-
cbrt	cbrt	-
ceil	ceil	-
copysign	copysign	-
erf	erf	-
erfc	erfc	-
exp2	exp2	-
expm1	expm1	-
fdim	fdim	-
floor	floor	-
fma	fma	-
fmax	fmax	-
fmin	fmin	-
fmod	fmod	-
frexp	frexp	-
hypot	hypot	-

Type-Generic Macro	<math.h> Functions	<complex.h> Functions
ilogb	ilogb	-
ldexp	ldexp	-
lgamma	lgamma	-
llrint	llrint	-
llround	llround	-
log10	log10	-
log1p	log1p	-
log2	log2	-
logb	logb	-
lrint	lrint	-
lround	lround	-
nearbyint	nearbyint	-
nextafter	nextafter	-
nexttoward	nexttoward	-
remainder	remainder	-
remquo	remquo	-
rint	rint	-
round	round	-
scalbn	scalbn	-
scalbln	scalbln	-
tgamma	tgamma	-
trunc	trunc	-
carg	-	carg
cimag	-	cimag
conj	-	conj
cproj	-	cproj
creal	-	creal

7.4.21 <wchar.h>

The following shows macros.

Type	Definition Name	Description
Macro	mbstate_t	Indicates the type for holding the necessary state of conversion between sequences of multibyte characters and wide characters.
	wint_t	Indicates the type for holding extended characters.
Constant (macro)	WEOF	Indicates the end-of-file.
Function	fwprintf	Converts the output format and outputs data to a stream.
	vfwprintf	Equivalent to **fwprintf** with the variable argument list replaced by **va_list**.
	swprintf	Converts the output format and writes data to an array of wide characters.
	vswprintf	Equivalent to **swprintf** with the variable argument list replaced by **va_list**.
	wprintf	Equivalent to **fwprintf** with **stdout** added as an argument before the specified arguments.
	vwprintf	Equivalent to **wprintf** with the variable argument list replaced by **va_list**.
	fwscanf	Inputs and converts data from the stream under control of the wide string and assigns it to an object.
	vfwscanf *<-lang=c99>*	Equivalent to **fwscanf** with the variable argument list replaced by **va_list**.
	swscanf	Converts data under control of the wide string and assigns it to an object.
	vswscanf *<-lang=c99>*	Equivalent to **swscanf** with the variable argument list replaced by **va_list**.
	wscanf	Equivalent to fwscanf with stdin added as an argument before the specified arguments.
	vwscanf *<-lang=c99>*	Equivalent to **wscanf** with the variable argument list replaced by **va_list**.
	fgetwc	Inputs a wide character as the **wchar_t** type and converts it to the **wint_t** type.
	fgetws	Stores a sequence of wide characters in an array.
	fputwc	Writes a wide character.
	fputws	Writes a wide string.
	fwide	Specifies the input/output unit.
	getwc	Equivalent to **fgetwc**.
	getwchar	Equivalent to **getwc** with **stdin** specified as an argument.
	putwc	Equivalent to **fputwc**.
	putwchar	Equivalent to **putwc** with **stdout** specified as the second argument.
	ungetwc	Returns a wide character to a stream.
	wcstod / wcstof / wcstold	These convert the initial part of a wide string to **double**, **float**, or **long double** representation.
	wcstol / wcstoll / wcstoul / wcstoull (wcstoll *<-lang=c99>*) (wcstoull *<-lang=c99>*)	These convert the initial part of a wide string to **long int**, **long long int**, **unsigned long int**, or **unsigned long long int** representation.

Type	Definition Name	Description
Function	wcscpy	Copies a wide string.
	wcsncpy	Copies **n** or fewer wide characters.
	wmemcpy	Copies **n** wide characters.
	wmemmove	Copies **n** wide characters.
	wcscat	Copies a wide string and appends it to the end of another wide string.
	wcsncat	Copies a wide string with **n** or fewer wide characters and appends it to the end of another wide character string.
	wcscmp	Compares two wide strings.
	wcsncmp	Compares two arrays with **n** or fewer wide characters.
	wmemcmp	Compares **n** wide characters.
	wcschr	Searches for a specified wide string in another wide string.
	wcscspn	Checks if a wide string contains another specified wide string.
	wcspbrk	Searches for the first occurrence of a specified wide string in another wide string.
	wcsrchr	Searches for the last occurrence of a specified wide character in a wide string.
	wcsspn	Calculates the length of the maximum initial segment of a wide string, which consists of specified wide characters.
	wcsstr	Searches for the first occurrence of a specified sequence of wide characters in a wide string.
	wcstok	Divides a wide string into a sequence of tokens delimited by a specified wide character.
	wmemchr	Searches for the first occurrence of a specified wide character within the first **n** wide characters in an object.
	wcslen	Calculates the length of a wide string.
	wmemset	Copies **n** wide characters.
	mbsinit	Checks if a specified object indicates the initial conversion state.
	mbrlen	Calculates the number of bytes in a multibyte character.

fwprintf

Outputs data to a stream input/output file according to the format.

[Format]

```
#include <stdio.h>
#include <wchar.h>
long fwprintf(FILE *restrict fp, const wchar_t *restrict control [, arg]...);
```

[Parameters]

fp File pointer
control Pointer to wide string indicating format
arg,... List of data to be output according to format

[Return values]

Normal: Number of wide strings converted and output
Abnormal: Negative value

[Remarks]

The **fwprintf** function is the wide-character version of the **fprintf** function.

vfwprintf

Outputs a variable parameter list to the specified stream input/output file according to a format.

[Format]

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
long vfwprintf(FILE *restrict fp, const char *restrict control, va_list arg);
```

[Parameters]

fp File pointer
control Pointer to wide string indicating format
arg Parameter list

[Return values]

Normal: Number of characters converted and output
Abnormal: Negative value

[Remarks]

The **vfwprintf** function is the wide-character version of the **vfprintf** function.

swprintf

Converts data according to a format and outputs it to the specified area.

[Format]

```
#include <stdio.h>
#include <wchar.h>
long swprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict control [, arg]...);
```

[Parameters]

s Pointer to storage area to which data is to be output
n Number of wide characters to be output
control Pointer to wide string indicating format
arg,... Data to be output according to format

[Return values]

Normal: Number of characters converted
Abnormal: When a representation format error occurs or writing n or morewide characters is requested: Negative value

[Remarks]

The **swprintf** function is the wide-character version of the **sprintf** function.

vswprintf

Outputs a variable parameter list to the specified storage area according to a format.

[Format]

```
#include <stdarg.h>
#include <wchar.h>
long vswprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict control, va_list arg);
```

[Parameters]

s Pointer to storage area to which data is to be output
n Number of wide characters to be output
control Pointer to wide string indicating format
arg Parameter list

[Return values]

Normal: Number of characters converted
Abnormal: Negative value

[Remarks]

The **vswprintf** function is the wide-character version of the **vsprintf** function.

wprintf

Converts data according to a format and outputs it to the standard output file (**stdout**).

[Format]

```
#include <stdio.h>
#include <wchar.h>
long wprintf(const wchar_t *restrict control [, arg]...);
```

[Parameters]

control Pointer to string indicating format
arg,... Data to be output according to format

[Return values]

Normal: Number of wide characters converted and output
Abnormal: Negative value

[Remarks]

The **wprintf** function is the wide-character version of **printf** function.

vwprintf

Outputs a variable parameter list to the standard output file (**stdout**) according to a format.

[Format]

#include <stdarg.h>
#include <wchar.h>
long vwprintf(const wchar_t *restrict control, va_list arg);

[Parameters]

control Pointer to wide string indicating format
arg Parameter list

[Return values]

Normal: Number of characters converted and output
Abnormal: Negative value

[Remarks]

The **vwprintf** function is the wide-character version of the **vprintf** function.

fwscanf

Inputs data from a stream input/output file and converts it according to a format.

[Format]

```
#include <stdio.h>
#include <wchar.h>
long fwscanf(FILE *restrict fp, const wchar_t *restrict control [, ptr]...);
```

[Parameters]

fp File pointer
control Pointer to wide string indicating format
ptr Pointer to storage area that stores input data

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: Input data ends before input data conversion is performed: **EOF**

[Remarks]

The **fwscanf** function is the wide-character version of the **fscanf** function.

vfwscanf

Inputs data from a stream input/output file and converts it according to a format.

[Format]

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
long vfwscanf(FILE *restrict fp, const wchar_t *restrict control, va_list arg);
```

[Parameters]

fp File pointer
control Pointer to wide string indicating format
arg Parameter list

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: Input data ends before input data conversion is performed: **EOF**

[Remarks]

The **vfwscanf** is the wide-character version of the **vfscanf** function.

swscanf

Inputs data from the specified storage area and converts it according to a format.

[Format]

```
#include <stdio.h>
#include <wchar.h>
long swscanf(const wchar_t *restrict s, const wchar_t *restrict control [, ptr]...);
```

[Parameters]

s Storage area containing data to be input
control Pointer to wide string indicating format
ptr,... Pointer to storage area that stores input and converted data

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: **EOF**

[Remarks]

The **swscanf** is the wide-character version of the **sscanf** function.

vswscanf

Inputs data from the specified storage area and converts it according to a format.

[Format]

```
#include <stdarg.h>
#include <wchar.h>
long vswscanf(const wchar_t *restrict s, const wchar_t *restrict control, va_list arg);
```

[Parameters]

s Storage area containing data to be input
control Pointer to wide string indicating format
arg Parameter list

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: **EOF**

wscanf

Inputs data from the standard input file (**stdin**) and converts it according to a format.

[Format]

```
#include <wchar.h>
long wscanf(const wchar_t *control [, ptr]...);
```

[Parameters]

control Pointer to wide string indicating format
ptr,... Pointer to storage area that stores input and converted data

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: **EOF**

[Remarks]

The **wscanf** function is the wide-character version of the **scanf** function.

vwscanf

Inputs data from the specified storage area and converts it according to a format.

[Format]

```
#include <stdarg.h>
#include <wchar.h>
long vwscanf(const wchar_t *restrict control, va_list arg);
```

[Parameters]

control Pointer to wide string indicating format
arg Parameter list

[Return values]

Normal: Number of data items successfully input and converted
Abnormal: Input data ends before input data conversion is performed: **EOF**

[Remarks]

The **vwscanf** function is provided to support wide-character format with the **vscanf** function.

fgetwc

Inputs one wide character from a stream input/output file.

[Format]

```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *fp);
```

[Parameters]

fp File pointer

[Return values]

Normal: End-of-file: **EOF**
Otherwise: Input wide character
Abnormal: **EOF**

[Remarks]

When a read error occurs, the error indicator for that file is set.
The **fgetwc** function is provided to support wide-character input to the **fgetc** function.

fgetws

Inputs a wide string from a stream input/output file.

[Format]

```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t *restrict s, long n, FILE *fp);
```

[Parameters]

s Pointer to storage area to which wide string is input
n Number of bytes of storage area to which wide string is input
fp File pointer

[Return values]

Normal: End-of-file: **NULL**
Otherwise: **s**
Abnormal: **NULL**

[Remarks]

The **fgetws** function is provided to support wide-character input to the **fgets** function.

fputwc

Outputs one wide character to a stream input/output file.

[Format]

```
#include <stdio.h>
#include <wchar.h>
wint_t fputwc(wchar_t c, FILE *fp);
```

[Parameters]

c Character to be output
fp File pointer

[Return values]

Normal: Output wide character
Abnormal: **EOF**

[Remarks]

When a write error occurs, the error indicator for that file is set.
The **fputwc** function is the wide-character version of the **fputc** function.

fputws

Outputs a wide string to a stream input/output file.

[Format]

```
#include <stdio.h>
#include <wchar.h>
long fputws(const wchar_t *restrict s, FILE *restrict fp);
```

[Parameters]

s Pointer to wide string to be output
fp File pointer

[Return values]

Normal: 0
Abnormal: **EOF**

[Remarks]

The **fputws** function is the wide-character version of the **fputs** function.

fwide

Specifies the input unit of a file.

[Format]

```
#include <stdio.h>
#include <wchar.h>
long fwide(FILE *fp, long mode);
```

[Parameters]

fp File pointer
mode Value indicating the input unit

[Return values]

A wide character is specified as the unit: Value greater than 0
A byte is specified as the unit: Value smaller than 0
No input/output unit is specified: 0

[Remarks]

The **fwide** function does not change the stream input/output unit that has already been determined.

getwc

Inputs one wide character from a stream input/output file.

[Format]

```
#include <stdio.h>
#include <wchar.h>
long getwc(FILE *fp);
```

[Parameters]

fp File pointer

[Return values]

Normal: End-of-file: **WEOF**
Otherwise: Input wide character
Abnormal: **EOF**

[Remarks]

When a read error occurs, the error indicator for that file is set.

The **getwc** function is equivalent to **fgetwc**, but **getwc** may evaluate **fp** two or more times because it is implemented as a macro. Accordingly, specify an expression without side effects for **fp**.

getwchar

Inputs one wide character from the standard input file (**stdin**).

[Format]

```
#include <wchar.h>
long getwchar(void);
```

[Return values]

Normal: End-of-file: **WEOF**
Otherwise: Input wide character
Abnormal: **EOF**

[Remarks]

When a read error occurs, the error indicator for that file is set.
The **getwchar** function is the wide-character version of the **getchar** function.

putwc

Outputs one wide character to a stream input/output file.

[Format]

```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t c, FILE *fp);
```

[Parameters]

c Wide character to be output
fp File pointer

[Return values]

Normal: Output wide character
Abnormal: **WEOF**

[Remarks]

When a write error occurs, the error indicator for that file is set.

The **putwc** function is equivalent to **fputwc**, but **putwc** may evaluate **fp** two or more times because it is implemented as a macro. Accordingly, specify an expression without side effects for **fp**.

putwchar

Outputs one wide character to the standard output file (**stdout**).

[Format]

```
#include <wchar.h>
wint_t putwchar(wchar_t c);
```

[Parameters]

c Wide character to be output

[Return values]

Normal: Output wide character
Abnormal: **WEOF**

[Remarks]

When a write error occurs, the error indicator for that file is set.
The **putwchar** function is the wide-character version of the **putchar** function.

ungetwc

Returns one wide character to a stream input/output file.

[Format]

```
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t c, FILE *fp);
```

[Parameters]

c Wide character to be returned
fp File pointer

[Return values]

Normal: Returned wide character
Abnormal: **WEOF**

[Remarks]

The **ungetwc** function is the wide-character version of the **ungetc** function.

wcstod / wcstof / wcstold

Converts the initial part of a wide string to a specified-type floating-point number.

[Format]

#include <wchar.h>
double wcstod(const wchar_t *restrict nptr, wchar_t **restrict endptr);
float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);
long double wcstold(const wchar_t *restrict nptr, wchar_t **restrict endptr);

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent a floating-point number

[Return Values]

Normal: f the string pointed by **nptr** begins with a character that does not represent a floating-point number: 0
If the string pointed by **nptr** begins with a character thatrepresents a floating-point number: Converted data as a specified-type floating-point number
Abnormal: If the converted data overflows: **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** with the same sign as that of the string before conversion
If the converted data underflows: 0

[Remarks]

If the converted result overflows or underflows, **errno** is set.
The **wcstod** function group is the wide-character version of the **strtod** function group.

wcstol / wcstoll / wcstoul / wcstoull

Converts the initial part of a wide string to a specified-type integer.

[Format]

```
#include <wchar.h>
long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);
long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);
unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);
unsigned long long int wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base;
```

[Parameters]

nptr Pointer to a number-representing string to be converted
endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer
base Radix of conversion (0 or 2 to 36)

[Return values]

Normal: f the string pointed by **nptr** begins with a character that does not represent an integer: 0
If the string pointed by **nptr** begins with a character that represents an integer: Converted data as a specified-type integer
Abnormal: If the converted data overflows: **LONG_MIN**, **LONG_MAX**, **LLONG_MIN**, **LLONG_MAX**, **ULONG_MAX**, or **ULLONG_MAX** depending on the sign of the string before conversion

[Remarks]

If the converted result overflows, **errno** is set.
The **wcstol** function group is the wide-character version of the **strtol** function group.

wcscpy

Copies the contents of a source wide string including the null character to a destination storage area.

[Format]

```
#include <wchar.h>
wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
```

[Parameters]

s1 Pointer to destination storage area
s2 Pointer to source string

[Return values]

s1 value

[Remarks]

The **wcscpy** function group is the wide-character version of the **strcpy** function group.

wcsncpy

Copies a source wide string of a specified length to a destination storage area.

[Format]

```
#include <wchar.h>
wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

[Parameters]

s1 Pointer to destination storage area
s2 Pointer to source string
n Number of characters to be copied

[Return values]

s1 value

[Remarks]

The **wcsncpy** function is the wide-character version of the **strncpy** function.

wmemcpy

Copies the contents of a source storage area of a specified length to a destination storage area.

[Format]

```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);
```

[Parameters]

s1 Pointer to destination storage area
s2 Pointer to source storage area
n Number of characters to be copied

[Return values]

s1 value

[Remarks]

The **wmemcpy** function is the wide-character version of the **memcpy** function.

wmemmove

Copies the specified size of the contents of a source area to a destination storage area. If part of the source storage area and the destination storage area overlap, data is copied to the destination storage area before the overlapped source storage area is overwritten. Therefore, correct copy is enabled.

[Format]

```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

[Parameters]

s1 Pointer to destination storage area
s2 Pointer to source storage area
n Number of characters to be copied

[Return values]

s1 value

[Remarks]

The **wmemmove** function is the wide-character version of the **memmove** function.

wcscat

Concatenates a string after another string.

[Format]

```
#include <wchar.h>
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
```

[Parameters]

s1 Pointer to the string after which another string is appended
s2 Pointer to the string to be appended after the other string

[Return values]

s1 value

[Remarks]

The **wcscat** function is the wide-character version of the **strcat** function.

wcsncat

Concatenates a string of a specified length after another string.

[Format]

```
#include <wchar.h>
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

[Parameters]

s1 Pointer to the string after which another string is appended
s2 Pointer to the string to be appended after the other string
n Number of characters to concatenate

[Return values]

s1 value

[Remarks]

The **wcsncat** function is the wide-character version of the **strncat** function.

wcscmp

Compares the contents of two strings specified.

[Format]

```
#include <wchar.h>
long wcscmp(const wchar_t *s1, const wchar_t *s2);
```

[Parameters]

s1 Pointer to the reference string to be compared
s2 Pointer to the string to compare to the reference

[Return values]

If string pointed by **s1** > string pointed by **s2**: Positive value
If string pointed by **s1** == string pointed by **s2**: 0
If string pointed by **s1** < string pointed by **s2**: Negative value

[Remarks]

The **wcscmp** function is the wide-character version of the **strcmp** function.

wcsncmp

Compares two strings specified up to a specified length.

[Format]

```
#include <wchar.h>
long wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

[Parameters]

s1 Pointer to the reference string to be compared
s2 Pointer to the string to compare to the reference
n Maximum number of characters to compare

[Return values]

If string pointed by **s1** > string pointed by **s2**: Positive value
If string pointed by **s1** == string pointed by **s2**: 0
If string pointed by **s1** < string pointed by **s2**: Negative value

[Remarks]

The **wcsncmp** function is the wide-character version of the **strncmp** function.

wmemcmp

Compares the contents of two storage areas specified.

[Format]

```
#include <wchar.h>
long wmemcmp(const wchar_t * s1, const wchar_t * s2, size_t n);
```

[Parameters]

s1 Pointer to the reference storage area to be compared
s2 Pointer to the storage area to compare to the reference
n Number of characters to compare

[Return values]

If storage area pointed by **s1** > storage area pointed by **s2**: Positive value
If storage area pointed by **s1** == storage area pointed by **s2**: 0
If storage area pointed by **s1** < storage area pointed by **s2**: Negative value

[Remarks]

The **wmemcmp** function is the wide-character version of the **memcmp** function.

wcschr

Searches a specified string for the first occurrence of a specified character.

[Format]

```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

[Parameters]

s Pointer to the string to be searched
c Character to search for

[Return values]

If the character is found: Pointer to the found character
If the character is not found: **NULL**

[Remarks]

The **wcschr** function is the wide-character version of the **strchr** function.

wcscspn

Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are not included in another string specified.

[Format]

```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

[Parameters]

s1 Pointer to the string to be checked
s2 Pointer to the string used to check **s1**

[Return values]

Number of characters at the beginning of the **s1** string that are not included in the **s2** string

[Remarks]

The **wcscspn** function is the wide-character version of the **strcspn** function.

wcspbrk

Searches a specified string for the first occurrence of the character that is included in another string specified.

[Format]

```
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

[Parameters]

s1 Pointer to the string to be searched
s2 Pointer to the string that indicates the characters to search **s1** for

[Return values]

If the character is found: Pointer to the found character
If the character is not found: **NULL**

[Remarks]

The **wcspbrk** function is the wide-character version of the **strpbrk** function.

wcsrchr

Searches a specified string for the last occurrence of a specified character.

[Format]

```
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

[Parameters]

s Pointer to the string to be searched
c Character to search for

[Return values]

If the character is found: Pointer to the found character
If the character is not found: **NULL**

wcsspn

Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are included in another string specified.

[Format]

```
#include <wchar.h>
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

[Parameters]

s1 Pointer to the string to be checked
s2 Pointer to the string used to check **s1**

[Return values]

Number of characters at the beginning of the **s1** string that are included in the **s2** string

[Remarks]

The **wcsspn** function is the wide-character version of the **strspn** function.

wcsstr

Searches a specified string for the first occurrence of another string specified.

[Format]

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

[Parameters]

s1 Pointer to the string to be searched
s2 Pointer to the string to search for

[Return values]

If the string is found: Pointer to the found string
If the string is not found: **NULL**

wcstok

Divides a specified string into some tokens.

[Format]

```
#include <wchar.h>
wchar_t* wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t ** restrict ptr);
```

[Parameters]

s1 Pointer to the string to be divided into some tokens
s2 Pointer to the string representing string-dividing characters
ptr Pointer to the string where search is to be started at the next function call

[Return values]

If division into tokens is successful: Pointer to the first token divided
If division into tokens is unsuccessful: **NULL**

[Remarks]

The **wcstok** function is the wide-character version of the **strtok** function.

To search the same string for the second or later time, set **s1** to **NULL** and **ptr** to the value returned by the previous function call to the same string.

wmemchr

Searches a specified storage area for the first occurrence of a specified character.

[Format]

```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

[Parameters]

s Pointer to the storage area to be searched
c Character to search for
n Number of characters to search

[Return values]

If the character is found: Pointer to the found character
If the character is not found: **NULL**

[Remarks]

The **wmemchr** function is the wide-character version of the **memchr** function.

wcslen

Calculates the length of a wide string except the terminating null wide character.

[Format]

```
#include <wchar.h>
size_t wcslen(const wchar_t *s);
```

[Parameters]

s Pointer to the wide string to check the length of

[Return values]

Number of characters in the wide string

[Remarks]

The **wcslen** function is the wide-character version of the **strlen** function.

wmemset

Sets a specified character a specified number of times at the beginning of a specified storage area.

[Format]

```
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

[Parameters]

s Pointer to storage area to set characters in
c Character to be set
n Number of characters to be set

[Return values]

Value of **s**

[Remarks]

The **wmemset** function is the wide-character version of the **memset** function.

mbsinit

Checks if a specified **mbstate_t** object indicates the initial conversion state.

[Format]

```
#include <wchar.h>
long mbsinit(const mbstate_t *ps);
```

[Parameters]

ps Pointer to **mbstate_t** object

[Return values]

Initial conversion state: Nonzero
Otherwise: 0

mbrlen

Calculates the number of bytes in a specified multibyte character.

[Format]

#include <wchar.h>
size_t mbrlen(const char * restrict s, size_t n, mbstate_t *restrict ps);

[Parameters]

s Pointer to multibyte string
n Maximum number of bytes to be checked for multibyte character
ps Pointer to **mbstate_t** object

[Return values]

0: A null wide character is detected in **n** or fewer bytes.
From 1 to n inclusive: A multibyte character is detected in **n** or fewer bytes.
(**size_t**)(–2): No complete multibyte character is detected in **n** bytes.
(**size_t**)(–1): An illegal multibyte sequence is detected.

7.5 EC++ Class Libraries

This section describes the specifications of the EC++ class libraries, which can be used as standard libraries in C++ programs. The class library types and corresponding standard include files are described. The specifications of each class library are given in accordance with the library configuration.

- Library types
  Table 7.15 shows the class library types and the corresponding standard include files.

Table 7.15 Class Library Types and Corresponding Standard Include Files

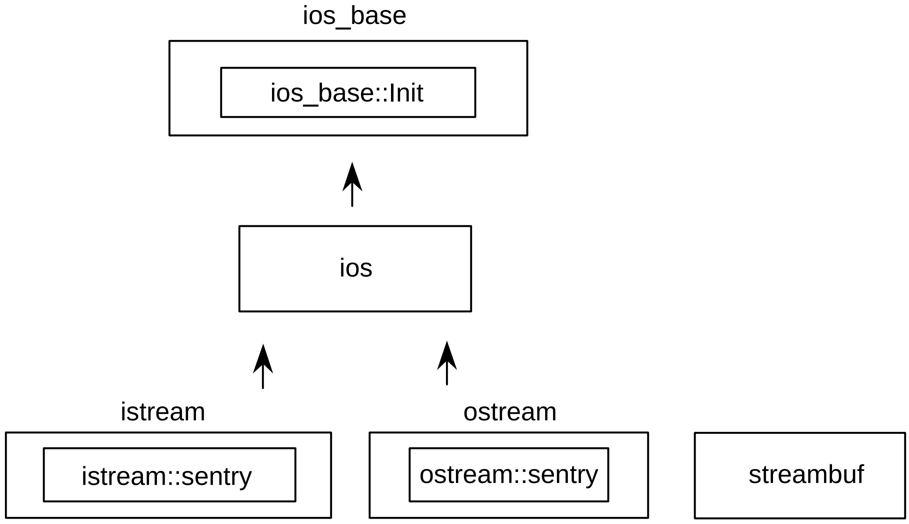
Library Type	Description	Standard Include Files
Stream input/output class library	Performs input/output processing	<ios>, <streambuf>, <istream>, <ostream>, <iostream>, <iomanip>
Memory management library	Performs memory allocation and deallocation	<new>
Complex number calculation class library	Performs calculation of complex number data	<complex>
String manipulation class library	Performs string manipulation	<string>

7.5.1 Stream Input/Output Class Library

The header files for stream input/output class libraries are as follows:

- <ios>
  Defines data members and function members that specify input/output formats and manage the input/output states. The **<ios>** header file also defines the **Init** and **ios_base** classes in addition to the ios class.
- <streambuf>
  Defines functions for the stream buffer.
- <istream>
  Defines input functions from the input stream.
- <ostream>
  Defines output functions to the output stream.
- <iostream>
  Defines input/output functions.
- <iomanip>
  Defines manipulators with parameters.

The following shows the inheritance relation of the above classes. An arrow (->) indicates that a derived class references a base class. The **streambuf** class has no inheritance relation.



The following types are used by stream input/output class libraries.

Type	Definition Name	Description
Type	streamoff	Defined as **long** type.
	streamsize	Defined as **size_t** type.
	int_type	Defined as **int** type.
	pos_type	Defined as **long** type.
	off_type	Defined as **long** type.

(a) ios_base::Init Class

Type	Definition Name	Description
Variable	init_cnt	Static data member that counts the number of stream input/output objects. The data must be initialized to 0 by a low-level interface.
Function	Init()	Constructor.
	~Init()	Destructor.

ios_base::Init::Init()

Constructor of class **Init**.
Increments **init_cnt**.

ios_base::Init::~Init()

Destructor of class **Init**.
Decrements **init_cnt**.

(b) ios_base Class

Type	Definition Name	Description
Type	fmtflags	Type that indicates the format control information.
	iostate	Type that indicates the stream buffer input/output state.
	openmode	Type that indicates the open mode of the file.
	seekdir	Type that indicates the seek state of the stream buffer.
Variable	fmtfl	Format flag.
	wide	Field width.
	prec	Precision (number of decimal point digits) at output.
	fillch	Fill character.
Function	void _ec2p_init_base()	Initializes the base class.
	void _ec2p_copy_base(ios_base&ios_base_dt)	Copies **ios_base_dt**.
	ios_base()	Constructor.
	~ios_base()	Destructor.
	fmtflags flags() const	References the format flag (**fmtfl**).
	fmtflags flags(fmtflags fmtflg)	Sets **fmtflg**&format flag (**fmtfl**) to the format flag (**fmtfl**).
	fmtflags setf(fmtflags fmtflg)	Sets **fmtflg** to format flag (**fmtfl**).
	fmtflags setf(fmtflags fmtflg, fmtflags mask)	Sets **mask**&**fmtflg** to the format flag (**fmtfl**).
	void unsetf(fmtflags mask)	Sets ~**mask**&format flag (**fmtfl**) to the format flag (**fmtfl**).
	char fill() const	References the fill character (**fillch**).
	char fill(char ch)	Sets **ch** as the fill character (**fillch**).
	int precision() const	References the precision (**prec**).
	streamsize precision(streamsize preci)	Sets **preci** as precision (**prec**).
	streamsize width() const	References the field width (**wide**).
	streamsize width(streamsize wd)	Sets **wd** as field width (**wide**).

ios_base::fmtflags

Defines the format control information relating to input/output processing.
The definition for each bit mask of **fmtflags** is as follows:

const ios_base::fmtflags ios_base::boolalpha	= 0x0000;
const ios_base::fmtflags ios_base::skipws	= 0x0001;
const ios_base::fmtflags ios_base::unitbuf	= 0x0002;
const ios_base::fmtflags ios_base::uppercase	= 0x0004;
const ios_base::fmtflags ios_base::showbase	= 0x0008;
const ios_base::fmtflags ios_base::showpoint	= 0x0010;
const ios_base::fmtflags ios_base::showpos	= 0x0020;
const ios_base::fmtflags ios_base::left	= 0x0040;
const ios_base::fmtflags ios_base::right	= 0x0080;
const ios_base::fmtflags ios_base::internal	= 0x0100;
const ios_base::fmtflags ios_base::adjustfield	= 0x01c0;
const ios_base::fmtflags ios_base::dec	= 0x0200;
const ios_base::fmtflags ios_base::oct	= 0x0400;
const ios_base::fmtflags ios_base::hex	= 0x0800;
const ios_base::fmtflags ios_base::basefield	= 0x0e00;
const ios_base::fmtflags ios_base::scientific	= 0x1000;
const ios_base::fmtflags ios_base::fixed	= 0x2000;
const ios_base::fmtflags ios_base::floatfield	= 0x3000;
const ios_base::fmtflags ios_base::_fmtmask	= 0x3fff;

ios_base::iostate

Defines the input/output state of the stream buffer.
The definition for each bit mask of **iostate** is as follows:

const ios_base::iostate ios_base::goodbit	= 0x0;
const ios_base::iostate ios_base::eofbit	= 0x1;
const ios_base::iostate ios_base::failbit	= 0x2;
const ios_base::iostate ios_base::badbit	= 0x4;
const ios_base::iostate ios_base::_statemask	= 0x7;

ios_base::openmode

Defines open mode of the file.
The definition for each bit mask of **openmode** is as follows:

const ios_base::openmode ios_base::in	= 0x01;	Opens the input file.
const ios_base::openmode ios_base::out	= 0x02;	Opens the output file.
const ios_base::openmode ios_base::ate	= 0x04;	Seeks for **eof** only once after the file has been opened.
const ios_base::openmode ios_base::app	= 0x08;	Seeks for **eof** each time the file is written to.
const ios_base::openmode ios_base::trunc	= 0x10;	Opens the file in overwrite mode.
const ios_base::openmode ios_base::binary	= 0x20;	Opens the file in binary mode.

ios_base::seekdir

Defines the seek state of the stream buffer.
Determines the position in a stream to continue the input/output of data.
The definition for each bit mask of **seekdir** is as follows:

const ios_base::seekdir ios_base::beg	= 0x0;
const ios_base::seekdir ios_base::cur	= 0x1;
const ios_base::seekdir ios_base::end	= 0x2;

void ios_base::_ec2p_init_base()

The initial settings are as follows:

```
fmtfl = skipws | dec;
wide = 0;
prec = 6;
fillch = ' ';
```

void ios_base::_ec2p_copy_base(ios_base& ios_base_dt)

Copies **ios_base_dt**.

ios_base::ios_base()

Constructor of class **ios_base**.
Calls **Init::Init()**.

ios_base::~ios_base()

Destructor of class **ios_base**.

ios_base::fmtflags ios_base::flags() const

References the format flag (**fmtfl**).
Return value: Format flag (**fmtfl**).

```
ios_base::fmtflags ios_base::flags(fmtflags fmtflg)
```

Sets **fmtflg**&format flag (**fmtfl**) to the format flag (**fmtfl**).
Return value: Format flag (**fmtfl**) before setting.

```
ios_base::fmtflags ios_base::setf(fmtflags fmtflg)
```

Sets **fmtflg** to the format flag (**fmtfl**).
Return value: Format flag (**fmtfl**) before setting.

```
ios_base::fmtflags ios_base::setf((fmtflags fmtflg, fmtflags mask)
```

Sets the **mask**&**fmtflg** value to the format flag (**fmtfl**).
Return value: Format flag (**fmtfl**) before setting.

```
void ios_base::unsetf(fmtflags mask)
```

Sets ~**mask**&format flag (**fmtfl**) to the format flag (**fmtfl**).

```
char ios_base::fill() const
```

References the fill character (**fillch**).
Return value: Fill character (**fillch**).

```
char ios_base::fill(char ch)
```

Sets **ch** as the fill character (**fillch**).
Return value: Fill character (**fillch**) before setting.

```
int ios_base::precision() const
```

References the precision (**prec**).
Return value: Precision (**prec**).

```
streamsize ios_base::precision(streamsize preci)
```

Sets **preci** as the precision (**prec**).
Return value: Precision (**prec**) before setting.

```
streamsize ios_base::width() const
```

References the field width (**wide**).
Return value: Field width (**wide**).

```
streamsize ios_base::width(streamsize wd)
```

Sets **wd** as the field width (**wide**).
Return value: Field width (**wide**) before setting.

(c) ios Class

Type	Definition Name	Description
Variable	sb	Pointer to the **streambuf** object.
	tiestr	Pointer to the **ostream** object.
	state	State flag of **streambuf**.
Function	ios()	Constructor.
	ios(streambuf* sbptr)	
	void init(streambuf* sbptr)	Performs initial setting.
	virtual ~ios()	Destructor.
	operator void*() const	Tests whether an error has been generated (!**state**&(**badbit** | **failbit**).
	bool operator!() const	Tests whether an error has been generated (**state**&(**badbit** | **failbit**)).
	iostate rdstate() const	References the state flag (**state**).
	void clear(iostate st = goodbit)	Clears the state flag (**state**) except for the specified state (**st**).
	void setstate(iostate st)	Specifies **st** as the state flag (**state**).
	bool good() const	Tests whether an error has been generated (**state**==**goodbit**).
	bool eof() const	Tests for the end of an input stream (**state**&**eofbit**).
	bool bad() const	Tests whether an error has been generated (**state**&**badbit**).
	bool fail() const	Tests whether the input text matches the requested pattern (**state**&(**badbit** | **failbit**)).
	ostream* tie() const	References the pointer to the **ostream** object (**tiestr**).
	ostream* tie(ostream* tstrptr)	Sets **tstrptr** as the pointer to the **ostream** object (**tiestr**).
	streambuf* rdbuf() const	References the pointer to the **streambuf** object (**sb**).
	streambuf* rdbuf(streambuf* sbptr)	Sets **sbptr** as the pointer to the **streambuf** object (**sb**).
	ios& copyfmt(const ios& rhs)	Copies the state flag (**state**) of **rhs**.

```
ios::ios()
```

Constructor of class **ios**.
Calls **init(0)** and sets the initial value to the member object.

```
ios::ios(streambuf* sbptr)
```

Constructor of class **ios**.
Calls **init(sbptr)** and sets the initial value to the member object.

```
void ios::init(streambuf* sbptr)
```

Sets **sbptr** to **sb**.
Sets **state** and **tiestr** to 0.

```
virtual ios::~ios()
```

Destructor of class **ios**.

```
ios::operator void*() const
```

Tests whether an error has been generated (!**state**&(**badbit** | **failbit**)).
Return value: An error has been generated: **false**
No error has been generated: **true**

```
bool ios::operator!() const
```

Tests whether an error has been generated (**state**&(**badbit** | **failbit**)).
Return value: An error has been generated: **true**
No error has been generated: **false**

```
iostate ios::rdstate() const
```

References the state flag (**state**).
Return value: State flag (**state**).

```
void ios::clear(iostate st = goodbit)
```

Clears the state flag (**state**) except for the specified state (**st**).
If the pointer to the **streambuf** object (**sb**) is 0, **badbit** is set to the state flag (**state**).

```
void ios::setstate(iostate st)
```

Sets st to the state flag (**state**).

```
bool ios::good() const
```

Tests whether an error has been generated (**state**==**goodbit**).
Return value: An error has been generated: **false**
No error has been generated: **true**

```
bool ios::eof() const
```

Tests for the end of the input stream (**state**&**eofbit**).
Return value: End of the input stream has been reached: **true**
End of the input stream has not been reached: **false**

```
bool ios::bad() const
```

Tests whether an error has been generated (**state**&**badbit**).
Return value: An error has been generated: **true**
No error has been generated: **false**

bool ios::fail() const

Tests whether the input text matches the requested pattern (**state**&(**badbit** | **failbit**)).
Return value: Does not match the requested pattern: **true**
Matches the requested pattern: **false**

ostream* ios::tie() const

References the pointer (**tiestr**) to the **ostream** object.
Return value: Pointer to the **ostream** object (**tiestr**).

ostream* ios::tie(ostream* tstrptr)

Sets **tstrptr** as the pointer (**tiestr**) to the **ostream** object.
Return value: Pointer to the **ostream** object (**tiestr**) before setting.

streambuf* ios::rdbuf() const

References the pointer to the **streambuf** object (**sb**).
Return value: Pointer to the **streambuf** object (**sb**).

streambuf* ios::rdbuf(streambuf* sbptr)

Sets **sbptr** as the pointer to the **streambuf** object (**sb**).
Return value: Pointer to the **streambuf** object (**sb**) before setting.

ios& ios::copyfmt(const ios& rhs)

Copies the state flag (**state**) of **rhs**.
Return value: ***this**

(d) ios Class Manipulators

Type	Definition Name	Description
Function	ios_base& showbase(ios_base& str)	Specifies the radix display prefix mode.
	ios_base& noshowbase(ios_base& str)	Clears the radix display prefix mode.
	ios_base& showpoint(ios_base& str)	Specifies the decimal-point generation mode.
	ios_base& noshowpoint(ios_base& str)	Clears the decimal-point generation mode.
	ios_base& showpos(ios_base& str)	Specifies the + sign generation mode.
	ios_base& noshowpos(ios_base& str)	Clears the + sign generation mode.
	ios_base& skipws(ios_base& str)	Specifies the space skipping mode.
	ios_base& noskipws (ios_base& str)	Clears the space skipping mode.
	ios_base& uppercase(ios_base& str)	Specifies the uppercase letter conversion mode.
	ios_base& nouppercase(ios_base& str)	Clears the uppercase letter conversion mode.
	ios_base& internal(ios_base& str)	Specifies the internal fill mode.
	ios_base& left(ios_base& str)	Specifies the left side fill mode.
	ios_base& right(ios_base& str)	Specifies the right side fill mode.
	ios_base& dec(ios_base& str)	Specifies the decimal mode.
	ios_base& hex(ios_base& str)	Specifies the hexadecimal mode.
	ios_base& oct(ios_base& str)	Specifies the octal mode.
	ios_base& fixed(ios_base& str)	Specifies the fixed-point mode.
	ios_base& scientific(ios_base& str)	Specifies the scientific description mode.

```
ios_base& showbase(ios_base& str)
```

Specifies an output mode of prefixing a radix at the beginning of data.
For a hexadecimal, 0x is prefixed. For a decimal, nothing is prefixed. For an octal, 0 is prefixed.
Return value: **str**

```
ios_base& noshowbase(ios_base& str)
```

Clears the output mode of prefixing a radix at the beginning of data.
Return value: **str**

```
ios_base& showpoint(ios_base& str)
```

Specifies the output mode of showing the decimal point.
If no precision is specified, six decimal-point (fraction) digits are displayed.
Return value: **str**

```
ios_base& noshowpoint(ios_base& str)
```

Clears the output mode of showing the decimal point.
Return value: **str**

ios_base& showpos(ios_base& str)

Specifies the output mode of generating the + sign (adds a + sign to a positive number).
Return value: **str**

ios_base& noshowpos(ios_base& str)

Clears the output mode of generating the + sign.
Return value: **str**

ios_base& skipws(ios_base& str)

Specifies the input mode of skipping spaces (skips consecutive spaces).
Return value: **str**

ios_base& noskipws(ios_base& str)

Clears the input mode of skipping spaces.
Return value: **str**

ios_base& uppercase(ios_base& str)

Specifies the output mode of converting letters to uppercases.
In hexadecimal, the radix will be uppercase letters 0X, and the numeric value letters will be uppercase letters.
The exponential representation of a floating-point value will also use uppercase letter E.
Return value: **str**

ios_base& nouppercase(ios_base& str)

Clears the output mode of converting letters to uppercases.
Return value: **str**

ios_base& internal(ios_base& str)

When data is output in the field width (**wide**) range, it is output in the order of
Sign and radix
Fill character (**fill**)
Numeric value
Return value: **str**

ios_base& left(ios_base& str)

When data is output in the field width (**wide**) range, it is aligned to the left.
Return value: **str**

ios_base& right(ios_base& str)

When data is output in the field width (**wide**) range, it is aligned to the right.
Return value: **str**

ios_base& dec(ios_base& str)

Specifies the conversion radix to the decimal mode.
Return value: **str**

```
ios_base& hex(ios_base& str)
```

Specifies the conversion radix to the hexadecimal mode.
Return value: **str**

```
ios_base& oct(ios_base& str)
```

Specifies the conversion radix to the octal mode.
Return value: **str**

```
ios_base& fixed(ios_base& str)
```

Specifies the fixed-point output mode.
Return value: **str**

```
ios_base& scientific(ios_base& str)
```

Specifies the scientific description output mode (exponential description).
Return value: **str**

(e) streambuf Class

Type	Definition Name	Description
Constant	eof	Indicates the end of the file.
Variable	_B_cnt_ptr	Pointer to the length of valid data in the buffer.
	B_beg_ptr	Pointer to the base pointer of the buffer.
	_B_len_ptr	Pointer to the length of the buffer.
	B_next_ptr	Pointer to the next position of the buffer from which data is to be read.
	B_end_ptr	Pointer to the end position of the buffer.
	B_beg_pptr	Pointer to the start position of the control buffer.
	B_next_pptr	Pointer to the next position of the buffer from which data is to be read.
	C_flg_ptr	Pointer to the input/output control flag of the file.

Type	Definition Name	Description
Function	char* _ec2p_getflag() const	References the pointer for the file input/output control flag.
	char*& _ec2p_gnptr()	References the pointer to the next position of the buffer from which data is to be read.
	char*& _ec2p_pnptr()	References the pointer to the next position of the buffer where data is to be written.
	void _ec2p_bcntplus()	Increments the valid data length of the buffer.
	void _ec2p_bcntminus()	Decrements the valid data length of the buffer.
	void _ec2p_setbPtr(char** begptr, char** curptr, long* cntptr, long* lenptr, char* flgptr)	Sets the pointers of **streambuf**.
	streambuf()	Constructor.
	virtual ~streambuf()	Destructor.
	streambuf* pubsetbuf(char* s, streamsize n)	Allocates the buffer for stream input/output. This function calls **setbuf (s,n)**[*1].
	pos_type pubseekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out)	Moves the position to read or write data in the input/output stream by using the method specified by **way**. This function calls **seekoff(off,way,which)**[*1].
	pos_type pubseekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out)	Calculates the offset from the beginning of the stream to the current position. This function calls **seekpos(sp,which)**[*1].
	int pubsync()	Flushes the output stream. This function calls **sync()**[*1].
	streamsize in_avail()	Calculates the offset from the end of the input stream to the current position.
	int_type snextc()	Reads the next character.
	int_type sbumpc()	Reads one character and sets the pointer to the next character.
	int_type sgetc()	Reads one character.
	int sgetn(char* s, streamsize n)	Reads **n** characters and sets them in the memory area specified by **s**.
	int_type sputbackc(char c)	Puts back the read position.
	int sungetc()	Puts back the read position.
	int sputc(char c)	Inserts character **c**.
	int_type sputn(const char* s, streamsize n)	Inserts **n** characters at the position pointed to by the amount specified by **s**.
	char* eback() const	Reads the start pointer of the input stream.

Type	Definition Name	Description
Function	char* gptr() const	Reads the next pointer of the input stream.
	char* egptr() const	Reads the end pointer of the input stream.
	void gbump(int n)	Moves the next pointer of the input stream by the amount specified by **n**.
	void setg(char* gbeg, char* gnext, char* gend)	Assigns each pointer of the input stream.
	char* pbase() const	Calculates the start pointer of the output stream.
	char* pptr() const	Calculates the next pointer of the output stream.
	char* epptr() const	Calculates the end pointer of the output stream.
	void pbump(int n)	Moves the next pointer of the output stream by the amount specified by **n**.
	void setp(char* pbeg, char* pend)	Assigns each pointer of the output stream.
	virtual streambuf* setbuf(char* s, streamsize n)*1	For each derived class, a defined operation is executed.
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))*1	Changes the stream position.
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))*1	Changes the stream position.
	virtual int sync()*1	Flushes the output stream.
	virtual int showmanyc()*1	Calculates the number of valid characters in the input stream.
	virtual streamsize xsgetn(char* s, streamsize n)	Sets **n** characters in the memory area specified by **s**.
	virtual int_type underflow()*1	Reads one character without moving the stream position.
	virtual int_type uflow()*1	Reads one character of the next pointer.
	virtual int_type pbackfail(int type c = eof)*1	Puts back the character specified by **c**.
	virtual streamsize xsputn(const char* s, streamsize n)	Inserts **n** characters in the position specified by **s**.
	virtual int_type overflow(int type c = eof)*1	Inserts character **c** in the output stream.

Notes 1. This class does not define the processing.

```
char* streambuf::_ec2p_getflag() const
```

References the pointer for the file input/output control flag.

```
char*& streambuf::_ec2p_gnptr()
```

References the pointer to the next position of the buffer from which data is to be read.

```
char*& streambuf::_ec2p_pnptr()
```

References the pointer to the next position of the buffer where data is to be written.

```
void streambuf::_ec2p_bcntplus()
```

Increments the valid data length of the buffer.

```
void streambuf::_ec2p_bcntminus()
```

Decrements the valid data length of the buffer.

```
void _ec2p_setbPtr(char** begptr, char** curptr, long* cntptr, long* lenptr, char* flgptr)
```

Sets the pointers of streambuf.

```
streambuf::streambuf()
```

Constructor.
The initial settings are as follows:
_B_cnt_ptr = B_beg_ptr = B_next_ptr = B_end_ptr = C_flg_ptr = _B_len_ptr = 0
B_beg_pptr = &B_beg_ptr
B_next_pptr = &B_next_ptr

```
virtual streambuf::~streambuf()
```

Destructor.

```
streambuf* streambuf::pubsetbuf(char* s, streamsize n)
```

Allocates the buffer for stream input/output.
This function calls **setbuf (s,n)**.
Return value: ***this**

```
pos_type streambuf::pubseekoff(off_type off, ios_base::seekdir way, ios_base::openmode which =
(ios_base::openmode)(ios_base::in | ios_base::out))
```

Moves the read or write position for the input/output stream by using the method specified by **way**.
This function calls **seekoff(off,way,which)**.
Return value: The stream position newly specified.

```
pos_type streambuf::pubseekpos(pos_type sp, ios_base::openmode which = (ios_base::open-
mode)(ios_base::in | ios_base::out))
```

Calculates the offset from the beginning of the stream to the current position.
Moves the current stream pointer by the amount specified by **sp**.
This function calls **seekpos(sp,which)**.
Return value: The offset from the beginning of the stream.

```
int streambuf::pubsync()
```

Flushes the output stream.
This function calls **sync()**.
Return value: 0

```
streamsize streambuf::in_avail()
```

Calculates the offset from the end of the input stream to the current position.
Return value:
If the position where data is read is valid: The offset from the end of the stream to the current position.
If the position where data is read is invalid: 0 (**showmanyc()** is called).

```
int_type streambuf::snextc()
```

Reads one character. If the character read is not **eof**, the next character is read.
Return value: If the character read is not **eof**: The character read
If the character read is **eof**: **eof**

```
int_type streambuf::sbumpc()
```

Reads one character and moves forward the pointer to the next.
Return value: If the position where data is read is valid: The character read
If the position where data is read is invalid: **eof**

```
int_type streambuf::sgetc()
```

Reads one character.
Return value: If the position where data is read is valid: The character read
If the position where data is read is invalid: **eof**

```
int streambuf::sgetn(char* s, streamsize n)
```

Sets **n** characters in the memory area specified by **s**. If an **eof** is found in the string read, setting is stopped.
Return value: The specified number of characters.

```
int_type streambuf::sputbackc(char c)
```

If the data read position is correct and the put back data of the position is the same as **c**, the read position is put back.
Return value: If the read position was put back: The value of **c**
If the read position was not put back: **eof**

int streambuf::sungetc()

If the data read position is correct, the read position is put back.
Return value: If the read position was put back: The value that was put back
If the read position was not put back: **eof**

int streambuf::sputc(char c)

Inserts character **c**.
Return value: If the write position is correct: The value of **c**
If the write position is incorrect: **eof**

int_type streambuf::sputn(const char* s, streamsize n)

Inserts **n** characters at the position specified by **s**.
If the buffer is smaller than **n**, the number of characters for the buffer is inserted.
Return value: The number of characters inserted.

char* streambuf::eback() const

Calculates the start pointer of the input stream.
Return value: Start pointer.

char* streambuf::gptr() const

Calculates the next pointer of the input stream.
Return value: Next pointer.

char* streambuf::egptr() const

Calculates the end pointer of the input stream.
Return value: End pointer.

void streambuf::gbump(int n)

Moves forward the next pointer of the input stream by the amount specified by **n**.

void streambuf::setg(char* gbeg, char* gnext, char* gend)

Sets each pointer of the input stream as follows:

```
*B_beg_pptr = gbeg;
*B_next_pptr = gnext;
 B_end_ptr = gend;
*_B_cnt_ptr = gend-gnext;
*_B_len_ptr = gend-gbeg;
```

char* streambuf::pbase() const

Calculates the start pointer of the output stream.
Return value: Start pointer.

char* streambuf::pptr() const

Calculates the next pointer of the output stream.
Return value: Next pointer.

char* streambuf::epptr() const

Calculates the end pointer of the output stream.
Return value: End pointer.

void streambuf::pbump(int n)

Moves forward the next pointer of the output stream by the amount specified by **n**.

void streambuf::setp(char* pbeg, char* pend)

The settings for each pointer of the output stream are as follows:
*B_beg_pptr = pbeg;
*B_next_pptr = pbeg;
B_end_ptr = pend;
*_B_cnt_ptr = pend-pbeg;
*_B_len_ptr = pend-pbeg;

virtual streambuf* streambuf::setbuf(char* s, streamsize n)

For each derived class from **streambuf**, a defined operation is executed.
Return value: ***this** (This class does not define the processing.)

virtual pos_type streambuf::seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode)(ios_base::in | ios_base::out))

Changes the stream position.
Return value: -1 (This class does not define the processing.)

virtual pos_type streambuf::seekpos(pos_type sp, ios_base::openmode = (ios_base::open-mode)(ios_base::in | ios_base::out))

Changes the stream position.
Return value: -1 (This class does not define the processing.)

virtual int streambuf::sync()

Flushes the output stream.
Return value: 0 (This class does not define the processing.)

virtual int streambuf::showmanyc()

Calculates the number of valid characters in the input stream.
Return value: 0 (This class does not define the processing.)

virtual streamsize streambuf::xsgetn(char* s, streamsize n)

Sets **n** characters in the memory area specified by **s**.
If the buffer is smaller than **n**, the number of characters for the buffer is inserted.
Return value: The number of characters input.

virtual int_type streambuf::underflow()

Reads one character without moving the stream position.
Return value: **eof** (This class does not define the processing.)

virtual int_type streambuf::uflow()

Reads one character of the next pointer.
Return value: **eof** (This class does not define the processing.)

virtual int_type streambuf::pbackfail(int_type c = eof)

Puts back the character specified by **c**.
Return value: **eof** (This class does not define the processing.)

virtual streamsize streambuf::xsputn(const char* s, streamsize n)

Inserts **n** characters specified by **s** in to the stream position.
If the buffer is smaller than **n**, the number of characters for the buffer is inserted.
Return value: The number of characters inserted.

virtual int_type streambuf::overflow(int_type c = eof)

Inserts character **c** in the output stream.
Return value: **eof** (This class does not define the processing.)

(f) istream::sentry Class

Type	Definition Name	Description
Variable	ok_	Whether the current state is input-enabled.
Function	sentry(istream& is, bool noskipws = false)	Constructor.
	~sentry()	Destructor.
	operator bool()	References **ok_**.

istream::sentry::sentry(istream& is, bool noskipws = _false)

Constructor of internal class **sentry**.
If **good()** is non-zero, enables input with or without a format.
If **tie()** is non-zero, flushes the related output stream.

istream::sentry::~sentry()

Destructor of internal class **sentry**.

istream::sentry::operator bool()

References **ok_**.
Return value: **ok_**

(g) istream Class

Type	Definition Name	Description
Variable	chcount	The number of characters extracted by the input function called last.
Function	int _ec2p_getistr(char* str, unsigned int dig, int mode)	Converts **str** with the radix specified by **dig**.
	istream(streambuf* sb)	Constructor.
	virtual ~istream()	Destructor.
	istream& operator>>(bool& n)	Stores the extracted characters in **n**.
	istream& operator>>(short& n)	
	istream& operator>>(unsigned short& n)	
	istream& operator>>(int& n)	
	istream& operator>>(unsigned int& n)	
	istream& operator>>(long& n)	
	istream& operator>>(unsigned long& n)	
	istream& operator>>(long long& n)	
	istream& operator>>(unsigned long long& n)	
	istream& operator>>(float& n)	
	istream& operator>>(double& n)	
	istream& operator>>(long double& n)	
	istream& operator>>(void*& p)	Converts the extracted characters to a pointer to **void** and stores them in **p**.
	istream& operator >>(streambuf* sb)	Extracts characters and stores them in the memory area specified by **sb**.
	streamsize gcount() const	Calculates **chcount** (number of characters extracted).
	int_type get()	Extracts a character
	istream& get(char& c)	Extracts characters and stores them in **c**.
	istream& get(signed char& c)	
	istream& get(unsigned char& c)	
	istream& get(char* s, streamsize n)	Extracts strings with size **n-1** and stores them in the memory area specified by **s**.
	istream& get(signed char* s, streamsize n)	
	istream& get(unsigned char* s, streamsize n)	

Type	Definition Name	Description
Function	istream& get(char* s, streamsize n, char delim)	Extracts strings with size **n-1** and stores them in the memory area specified by **s**. If **delim** is found in the string, input is stopped.
	istream& get(signed char* s, streamsize n, char delim)	
	istream& get(unsigned char* s, streamsize n, char delim)	
	istream& get(streambuf& sb)	Extracts strings and stores them in the memory area specified by **sb**.
	istream& get(streambuf& sb, char delim)	Extracts strings and stores them in the memory area specified by **sb**. If **delim** is found in the string, input is stopped.
	istream& getline(char* s, streamsize n)	Extracts strings with size **n-1** and stores them in the memory area specified by **s**.
	istream& getline(signed char* s, streamsize n)	
	istream& getline(unsigned char* s, streamsize n)	
	istream& getline(char* s, streamsize n, char delim)	Extracts strings with size **n-1** and stores them in the memory area specified by **s**. If **delim** is found in the string, input is stopped.
	istream& getline(signed char* s, streamsize n, char delim)	
	istream& getline(unsigned char* s, streamsize n, char delim)	
	istream& ignore(streamsize n = 1, int_type delim = streambuf::eof)	Skips reading the number of characters specified by **n**. If **delim** is found in the string, skipping is stopped.
	int_type peek()	Seeks for input characters that can be acquired next.
	istream& read(char* s, streamsize n)	Extracts strings with size **n** and stores them in the memory area specified by **s**.
	istream& read(signed char* s, streamsize n)	
	istream& read(unsigned char* s, streamsize n)	
	streamsize readsome(char* s, streamsize n)	Extracts strings with size **n** and stores them in the memory area specified by **s**.
	streamsize readsome(signed char* s, streamsize n)	
	streamsize readsome(unsigned char* s, streamsize n)	

Type	Definition Name	Description
Function	istream& putback(char c)	Puts back a character to the input stream.
	istream& unget()	Puts back the position of the input stream.
	int sync()	Checks the existence of the input stream. This function calls **streambuf::pubsync()**.
	pos_type tellg()	Finds the input stream position. This function calls **streambuf::pub-seekoff(0,cur,in)**.
	istream& seekg(pos_type pos)	Moves the current stream pointer by the amount specified by **pos**. This function calls **streambuf::pubseek-pos(pos)**.
	istream& seekg(off_type off, ios_base::seekdir dir)	Moves the position to read the input stream by using the method specified by **dir**. This function calls **streambuf::pub-seekoff(off,dir)**.

int istream::_ec2p_getistr(char* str, unsigned int dig, int mode)

Converts **str** to the radix specified by **dig**.
Return value: The converted radix.

istream::istream(streambuf* sb)

Constructor of class **istream**.
Calls **ios::init(sb)**.
Specifies **chcount**=0.

virtual istream::~istream()

Destructor of class **istream**.

istream& istream::operator>>(bool& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(short& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(unsigned short& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(int& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(unsigned int& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(long& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(unsigned long& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(long long& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(unsigned long long& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(float& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(double& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(long double& n)

Stores the extracted characters in **n**.
Return value: ***this**

istream& istream::operator>>(void*& p)

Converts the extracted characters to a **void*** type and stores them in the memory specified by **p**.
Return value: ***this**

istream& istream::operator>>(streambuf* sb)

Extracts characters and stores them in the memory area specified by **sb**.
If there are no extracted characters, **setstate(failbit)** is called.
Return value: ***this**

streamsize istream::gcount() const

References **chcount** (number of extracted characters).
Return value: **chcount**

int_type istream::get()

Extracts characters.
Return value: If characters are extracted: Extracted characters.
If no characters are extracted: Calls **setstate(failbit)** and becomes **streambuf::eof**.

istream& istream::get(char& c)

Extracts characters and stores them in **c**. If the extracted character is **streambuf::eof**, **failbit** is set.
Return value: ***this**

istream& istream::get(signed char& c)

Extracts characters and stores them in **c**. If the extracted character is **streambuf::eof**, **failbit** is set.
Return value: ***this**

istream& istream::get(unsigned char& c)

Extracts characters and stores them in **c**. If the extracted character is **streambuf::eof**, **failbit** is set.
Return value: ***this**

istream& istream::get(char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::get(signed char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::get(unsigned char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::get(char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If **delim** is found in the string, input is stopped.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::get(signed char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If **delim** is found in the string, input is stopped.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::get(unsigned char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If **delim** is found in the string, input is stopped.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::get(streambuf& sb)

Extracts a string and stores it in the memory area specified by **sb**.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::get(streambuf& sb, char delim)

Extracts a string and stores it in the memory area specified by **sb**.
If **delim** is found in the string, input is stopped.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::getline(char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::getline(signed char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::getline(unsigned char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::getline(char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If character **delim** is found, input is stopped.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::getline(signed char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If character **delim** is found, input is stopped.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::getline(unsigned char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**.
If character **delim** is found, input is stopped.
If **ok_**==**false** or no character has been extracted, **failbit** is set.
Return value: ***this**

istream& istream::ignore(streamsize n = 1, int_type delim = streambuf::eof)

Skips reading the number of characters specified by **n**.
If character **delim** is found, skipping is stopped.
Return value: ***this**

int_type istream::peek()

Seeks input characters that will be available next.
Return value: If **ok_**==**false**: **streambuf::eof**
If **ok_**!=**false**: **rdbuf()->sgetc()**

istream& istream::read(char* s, streamsize n)

If **ok_**!=**false**, extracts a string with size **n** and stores it in the memory area specified by **s**. If the number of extracted characters does not match with the number of **n**, **eofbit** is set.
Return value: ***this**

istream& istream::read(signed char* s, streamsize n)

If **ok_**!=**false**, extracts a string with size **n** and stores it in the memory area specified by **s**. If the number of extracted characters does not match with the number of **n**, **eofbit** is set.
Return value: ***this**

istream& istream::read(unsigned char* s, streamsize n)

If **ok_**!=**false**, extracts a string with size **n** and stores it in the memory area specified by **s**. If the number of extracted characters does not match with the number of **n**, **eofbit** is set.
Return value: ***this**

streamsize istream::readsome(char* s, streamsize n)

Extracts a string with size **n** and stores it in the memory area specified by **s**.
If the number of characters exceeds the stream size, only the number of characters equal to the stream size is stored.
Return value: The number of extracted characters.

streamsize istream::readsome(signed char* s, streamsize n)

Extracts a string with size **n** and stores it in the memory area specified by **s**.
If the number of characters exceeds the stream size, only the number of characters equal to the stream size is stored.
Return value: The number of extracted characters.

streamsize istream::readsome(unsigned char* s, streamsize n)

Extracts a string with size **n** and stores it in the memory area specified by **s**.
If the number of characters exceeds the stream size, only the number of characters equal to the stream size is stored.
Return value: The number of extracted characters.

istream& istream::putback(char c)

Puts back character **c** to the input stream.
If the characters put back are **streambuf::eof**, **badbit** is set.
Return value: ***this**

istream& istream::unget()

Puts back the pointer of the input stream by one.
If the extracted characters are **streambuf::eof**, **badbit** is set.
Return value: ***this**

int istream::sync()

Checks for an input stream.
This function calls **streambuf::pubsync()**.
Return value: If there is no input stream: **streambuf::eof**
If there is an input stream: 0

pos_type istream::tellg()

Checks for the position of the input stream.
This function calls **streambuf::pubseekoff(0,cur,in)**.
Return value: Offset from the beginning of the stream
If an error occurs during the input processing, -1 is returned.

istream& istream::seekg(pos_type pos)

Moves the current stream pointer by the amount specified by **pos**.
This function calls **streambuf::pubseekpos(pos)**.
Return value: ***this**

istream& istream::seekg(off_type off, ios_base::seekdir dir)

Moves the position to read the input stream using the method specified by **dir**.
This function calls **streambuf::pubseekoff(off,dir)**. If an error occurs during the input processing, this processing is not performed.
Return value: ***this**

(h) istream Class Manipulator

Type	Definition Name	Description
Function	istream& ws(istream& is)	Skips reading the spaces.

istream& ws(istream& is)

Skips reading white spaces.
Return value: **is**

(i) istream Non-Member Function

Type	Definition Name	Description
Function	istream& operator>>(istream& in, char* s)	Extracts a string and stores it in the memory area specified by **s**.
	istream& operator>>(istream& in, signed char* s)	
	istream& operator>>(istream& in, unsigned char* s)	
	istream& operator>>(istream& in, char& c)	Extracts a character and stores it in **c**.
	istream& operator>>(istream& in, signed char& c)	
	istream& operator>>(istream& in, unsigned char& c)	

istream& operator>>(istream& in, char* s)

Extracts a string and stores it in the memory area specified by **s**.
Processing is stopped if
the number of characters stored is equal to field width – 1
streambuf::eof is found in the input stream
the next available character **c** satisfies **isspace(c)**==**1**
If no characters are stored, **failbit** is set.
Return value: **in**

istream& operator>>(istream& in, signed char* s)

Extracts a string and stores it in the memory area specified by **s**.
Processing is stopped if
the number of characters stored is equal to field width – 1
streambuf::eof is found in the input stream
the next available character **c** satisfies **isspace(c)**==**1**
If no characters are stored, **failbit** is set.
Return value: **in**

istream& operator>>(istream& in, unsigned char* s)

Extracts a string and stores it in the memory area specified by **s**.
Processing is stopped if
the number of characters stored is equal to field width – 1
streambuf::eof is found in the input stream
the next available character **c** satisfies **isspace(c)**==**1**
If no characters are stored, **failbit** is set.
Return value: **in**

istream& operator>>(istream& in, char& c)

Extracts a character and stores it in **c**. If no character is stored, **failbit** is set.
Return value: **in**

istream& operator>>(istream& in, signed char& c)

Extracts a character and stores it in **c**. If no character is stored, **failbit** is set.
Return value: **in**

istream& operator>>(istream& in, unsigned char& c)

Extracts a character and stores it in **c**. If no character is stored, **failbit** is set.
Return value: **in**

(j) ostream::sentry Class

Type	Definition Name	Description
Variable	ok_	Whether or not the current state allows output.
	_ _ec2p_os	Pointer to the **ostream** object.
Function	sentry(ostream& os)	Constructor.
	~sentry()	Destructor.
	operator bool()	References **ok_**.

ostream::sentry::sentry(ostream& os)

Constructor of the internal class **sentry**.
If **good()** is non-zero and **tie()** is non-zero, **flush()** is called.
Specifies **os** to _ _**ec2p_os**.

ostream::sentry::~sentry()

Destructor of internal class **sentry**.
If (_ _**ec2p_os->flags()** & **ios_base::unitbuf**) is true, **flush()** is called.

ostream::sentry::operator bool()

References **ok_**.
Return value: **ok_**

(k) ostream Class

Type	Definition Name	Description
Function	ostream(streambuf* sbptr)	Constructor.
	virtual ~ostream()	Destructor.
	ostream& operator<<(bool n)	Inserts **n** in the output stream.
	ostream& operator<<(short n)	
	ostream& operator<<(unsigned short n)	
	ostream& operator<<(int n)	
	ostream& operator<<(unsigned int n)	
	ostream& operator<<(long n)	
	ostream& operator<<(unsigned long n)	
	ostream& operator<<(long long n)	
	ostream& operator<<(unsigned long long n)	
	ostream& operator<<(float n)	
	ostream& operator<<(double n)	
	ostream& operator<<(long double n)	
	ostream& operator<<(void* n)	
	ostream& operator<<(streambuf* sbptr)	Inserts the output string of **sbptr** into the output stream.
	ostream& put(char c)	Inserts character **c** into the output stream.
	ostream& write(const char* s, streamsize n)	Inserts **n** characters from **s** into the output stream.
	ostream& write(const signed char* s, streamsize n)	
	ostream& write(const unsigned char* s, streamsize n)	
	ostream& flush()	Flushes the output stream. This function calls **streambuf::pubsync()**.
	pos_type tellp()	Calculates the current write position. This function calls **streambuf::pub-seekoff(0,cur,out)**.
	ostream& seekp(pos_type pos)	Calculates the offset from the beginning of the stream to the current position. Moves the current stream pointer by the amount specified by **pos**. This function calls **streambuf::pubseek-pos(pos)**.
	ostream& seekp(off_type off, seekdir dir)	Moves the stream write position by the amount specified by **off**, from **dir**. This function calls **streambuf::pub-seekoff(off,dir)**.

```
ostream::ostream(streambuf* sbptr)
```

Constructor.
Calls **ios(sbptr)**.

```
virtual ostream::~ostream()
```

Destructor.

```
ostream& ostream::operator<<(bool n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(short n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(unsigned short n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(int n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(unsigned int n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(long n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(unsigned long n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(long long n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(unsigned long long n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(float n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(double n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(long double n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(void* n)
```

If **sentry::ok_==true**, **n** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::operator<<(streambuf* sbptr)
```

If **sentry::ok_==true**, the output string of **sbptr** is inserted into the output stream.
If **sentry::ok_==false**, **failbit** is set.
Return value: ***this**

```
ostream& ostream::put(char c)
```

If (**sentry::ok_==true**) and (**rdbuf()->sputc(c)**!=**streambuf::eof**), **c** is inserted into the output stream.
Otherwise **badbit** is set.
Return value: ***this**

```
ostream& ostream::write(const char* s, streamsize n)
```

If (**sentry::ok_==true**) and (**rdbuf()->sputn(s, n)**==**n**), **n** characters specified by **s** are inserted into the output stream.
Otherwise **badbit** is set.
Return value: ***this**

ostream& ostream::write(const signed char* s, streamsize n)

If (**sentry::ok_==true**) and (**rdbuf()->sputn(s, n)**==**n**), **n** characters specified by **s** are inserted into the output stream.
Otherwise **badbit** is set.
Return value: ***this**

ostream& ostream::write(const unsigned char* s, streamsize n)

If (**sentry::ok_==true**) and (**rdbuf()->sputn(s, n)**==**n**), **n** characters specified by **s** are inserted into the output stream.
Otherwise **badbit** is set.
Return value: ***this**

ostream& ostream::flush()

Flushes the output stream.
This function calls **streambuf::pubsync()**.
Return value: ***this**

pos_type ostream::tellp()

Calculates the current write position.
This function calls **streambuf::pubseekoff(0,cur,out)**.
Return value: The current stream position
If an error occurs during processing, -1 is returned.

ostream& ostream::seekp(pos_type pos)

If no error occurs, the offset from the beginning of the stream to the current position is calculated.
Moves the current stream pointer by the amount specified by **pos**.
This function calls **streambuf::pubseekpos(pos)**.
Return value: ***this**

ostream& ostream::seekp(off_type off, seekdir dir)

If no error occurs, the stream write position is moved by the amount specified by **off**, from **dir**.
This function calls **streambuf::pubseekoff(off,dir)**.
Return value: ***this**

(l) ostream Class Manipulator

Type	Definition Name	Description
Function	ostream& endl(ostream& os)	Inserts a new line and flushes the output stream.
	ostream& ends(ostream& os)	Inserts a **NULL** code.
	ostream& flush(ostream& os)	Flushes the output stream.

ostream& endl(ostream& os)

Inserts a new line code and flushes the output stream.
This function calls **flush()**.
Return value: **os**

```
ostream& ends(ostream& os)
```

Inserts a **NULL** code into the output line.
Return value: **os**

```
ostream& flush(ostream& os)
```

Flushes the output stream.
This function calls **streambuf::sync()**.
Return value: **os**

(m) ostream Non-Member Function

Type	Definition Name	Description
Function	ostream& operator<<(ostream& os, char s)	Inserts **s** into the output stream.
	ostream& operator<<(ostream& os, signed char s)	
	ostream& operator<<(ostream& os, unsigned char s)	
	ostream& operator<<(ostream& os, const char* s)	
	ostream& operator<<(ostream& os, const signed char* s)	
	ostream& operator<<(ostream& os, const unsigned char* s)	

```
ostream& operator<<(ostream& os, char s)
```

If (**sentry::ok_==true**) and an error does not occur, **s** is inserted into the output stream. Otherwise **failbit** is set.
Return value: **os**

```
ostream& operator<<(ostream& os, signed char s)
```

If (**sentry::ok_==true**) and an error does not occur, **s** is inserted into the output stream. Otherwise **failbit** is set.
Return value: **os**

```
ostream& operator<<(ostream& os, unsigned char s)
```

If (**sentry::ok_==true**) and an error does not occur, **s** is inserted into the output stream. Otherwise **failbit** is set.
Return value: **os**

```
ostream& operator<<(ostream& os, const char* s)
```

If (**sentry::ok_==true**) and an error does not occur, **s** is inserted into the output stream. Otherwise **failbit** is set.
Return value: **os**

```
ostream& operator<<(ostream& os, const signed char* s)
```

If (**sentry::ok_==true**) and an error does not occur, **s** is inserted into the output stream. Otherwise **failbit** is set.
Return value: **os**

```
ostream& operator<<(ostream& os, const unsigned char* s)
```

If (**sentry::ok_==true**) and an error does not occur, **s** is inserted into the output stream. Otherwise **failbit** is set.
Return value: **os**

(n) smanip Class Manipulator

Type	Definition Name	Description
Function	smanip resetiosflags(ios_base::fmtflags mask)	Clears the flag specified by the **mask** value.
	smanip setiosflags(ios_base::fmtflags mask)	Specifies the format flag (**fmtfl**).
	smanip setbase(int base)	Specifies the radix used at output.
	smanip setfill(char c)	Specifies the fill character (**fillch**).
	smanip setprecision(int n)	Specifies the precision (**prec**).
	smanip setw(int n)	Specifies the field width (**wide**).

smanip resetiosflags(ios_base::fmtflags mask)

Clears the flag specified by the **mask** value.
Return value: Target object of input/output.

smanip setiosflags(ios_base::fmtflags mask)

Specifies the format flag (**fmtfl**).
Return value: Target object of input/output.

smanip setbase(int base)

Specifies the radix used at output.
Return value: Target object of input/output.

smanip setfill(char c)

Specifies the fill character (**fillch**).
Return value: Target object of input/output.

smanip setprecision(int n)

Specifies the precision (**prec**).
Return value: Target object of input/output.

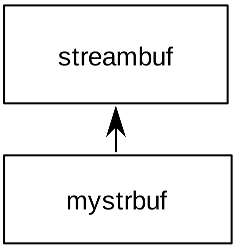
smanip setw(int n)

Specifies the field width (**wide**).
Return value: Target object of input/output.

(o) Example of Using EC++ Input/Output Libraries

The input/output stream can be used if a pointer to an object of the **mystrbuf** class is used instead of **streambuf** at the initialization of the **istream** and **ostream** objects.

The following shows the inheritance relationship of the above classes. An arrow (->) indicates that a derived class references a base class.



Type	Definition Name	Description
Variable	_file_Ptr	File pointer.
Function	mystrbuf()	Constructor. Initializes the **streambuf** buffer.
	mystrbuf(void* ptr)	
	virtual ~mystrbuf()	Destructor.
	void* myfptr() const	Returns a pointer to the **FILE** type structure.
	mystrbuf* open(const char* filename, int mode)	Specifies the file name and mode, and opens the file.
	mystrbuf* close()	Closes the file.
	virtual streambuf* setbuf(char* s, streamsize n)	Allocates the stream input/output buffer.
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	Changes the position of the stream pointer.
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	Changes the position of the stream pointer.
	virtual int sync()	Flushes the stream.
	virtual int showmanyc()	Returns the number of valid characters in the input stream.
	virtual int_type underflow()	Reads one character without moving the stream position.
	virtual int_type pbackfail(int type c = streambuf::eof)	Puts back the character specified by **c**.
	virtual int_type overflow(int type c = streambuf::eof)	Inserts the character specified by **c**.
	void _Init(_f_type* fp)	Initialization.

```
<Example>
#include <istream>
#include <ostream>
#include <mystrbuf>
#include <string>
#include <new>
#include <stdio.h>
void main(void)
{
    mystrbuf myfin(stdin);
    mystrbuf myfout(stdout);
    istream mycin(&myfin);
    ostream mycout(&myfout);

    int i;
    short s;
    long l;
    char c;
    string str;

    mycin >> i >> s >> l >> c >> str;
    mycout << "This is EC++ Library." << endl << i << s << l << c << str << endl;

    return;
}
```

7.5.2 Memory Management Library

The header file for the memory management library is as follows:

- <new>
  Defines the memory allocation/deallocation function.

By setting an exception handling function address to the **_ec2p_new_handler** variable, exception handling can be executed if memory allocation fails. The **_ec2p_new_handler** is a **static** variable and the initial value is **NULL**. If this handler is used, reentrance will be lost.

Operations required for the exception handling function:

- Creates an allocatable area and returns the area.
- Operations are not prescribed for cases where an area cannot be created.

Type	Definition Name	Description
Type	new_handler	Pointer type to the function that returns a **void** type.
Variable	_ec2p_new_handler	Pointer to an exception handling function.

Type	Definition Name	Description
Function	void* operator new(size_t size)	Allocates a memory area with a size specified by **size.**
	void* operator new[ ](size_t size)	Allocates an array area with a size specified by **size.**
	void* operator new(size_t size, void* ptr)	Allocates the area specified by **ptr** as the memory area.
	void* operator new[ ](size_t size, void* ptr)	Allocates the area specified by **ptr** as the array area.
	void operator delete(void* ptr)	Deallocates the memory area.
	void operator delete[ ](void* ptr)	Deallocates the array area.
	new_handler set_new_handler(new_handler new_P)	Sets the exception handling function address (**new_P**) in **_ec2p_new_handler**.

```
void* operator new(size_t size)
```

Allocates a memory area with the size specified by **size**.
If memory allocation fails and when **new_handler** is set, **new_handler** is called.
Return value: If memory allocation succeeds: Pointer to **void** type
If memory allocation fails: **NULL**

```
void* operator new[ ](size_t size)
```

Allocates an array area with the size specified by **size**.
If memory allocation fails and when **new_handler** is set, **new_handler** is called.
Return value: If memory allocation succeeds: Pointer to **void** type
If memory allocation fails: **NULL**

```
void* operator new(size_t size, void* ptr)
```

Allocates the area specified by **ptr** as the storage area.
Return value: **ptr**

```
void* operator new[ ](size_t size, void* ptr)
```

Allocates the area specified by **ptr** as the array area.
Return value: **ptr**

```
void operator delete(void* ptr)
```

Deallocates the storage area specified by **ptr**.
If **ptr** is **NULL**, no operation will be performed.

```
void operator delete[ ](void* ptr)
```

Deallocates the array area specified by **ptr**.
If **ptr** is **NULL**, no operation will be performed.

```
new_handler set_new_handler(new_handler new_P)
```

Sets **new_P** to **_ec2p_new_handler**.
Return value: **_ec2p_new_handler**.

7.5.3 Complex Number Calculation Class Library

The header file for the complex number calculation class library is as follows:

- <complex>
  Defines the **float_complex** and **double_complex** classes.

These classes have no derivation.

(a) float_complex Class

Type	Definition Name	Description
Type	value_type	**float** type
Variable	_re	Defines the real part of **float** precision.
	_im	Defines the imaginary part of **float** precision.
Function	float_complex(float re = 0.0f, float im = 0.0f)	Constructor.
	float_complex(const double_complex& rhs)	
	float real() const	Acquires the real part (**_re**).
	float imag() const	Acquires the imaginary part (**_im**).
	float_complex& operator=(float rhs)	Copies **rhs** to the real part.. 0.0f is assigned to the imaginary part.
	float_complex& operator+=(float rhs)	Adds **rhs** to the real part and stores the sum in ***this**.
	float_complex& operator-=(float rhs)	Subtracts **rhs** from the real part and stores the difference in ***this**.
	float_complex& operator*=(float rhs)	Multiplies ***this** by **rhs** and stores the product in ***this**.
	float_complex& operator/=(float rhs)	Divides ***this** by **rhs** and stores the quotient in ***this**.
	float_complex& operator=(const float_complex& rhs)	Copies **rhs**.
	float_complex& operator+=(const float_complex& rhs)	Adds **rhs** to ***this** and stores the sum in ***this**.
	float_complex& operator-=(const float_complex& rhs)	Subtracts **rhs** from ***this** and stores the difference in ***this**.
	float_complex& operator*=(const float_complex& rhs)	Multiplies ***this** by **rhs** and stores the product in ***this**.
	float_complex& operator/=(const float_complex& rhs)	Divides ***this** by **rhs** and stores the quotient in ***this**.

```
float_complex::float_complex(float re = 0.0f, float im = 0.0f)
```

Constructor of class **float_complex**.
The initial settings are as follows:
_re = re;
_im = im;

```
float_ complex::float_complex(const double_complex& rhs)
```

Constructor of class **float_complex**.
The initial settings are as follows:
_re = (float)rhs.real();
_im = (float)rhs.imag();

```
float float_complex::real() const
```

Acquires the real part.
Return value: **this->_re**

```
float float_complex::imag() const
```

Acquires the imaginary part.
Return value: **this->_im**

```
float_complex& float_complex::operator=(float rhs)
```

Copies **rhs** to the real part (**_re**).
0.0f is assigned to the imaginary part (**_im**).
Return value: ***this**

```
float_complex& float_complex::operator+=(float rhs)
```

Adds **rhs** to the real part (**_re**) and stores the result in the real part (**_re**).
The value of the imaginary part (**_im**) does not change.
Return value: ***this**

```
float_complex& float_complex::operator-=(float rhs)
```

Subtracts **rhs** from the real part (**_re**) and stores the result in the real part (**_re**).
The value of the imaginary part (**_im**) does not change.
Return value: ***this**

```
float_complex& float_complex::operator*=(float rhs)
```

Multiplies ***this** by **rhs** and stores the result in ***this**.
(_re=_re*rhs, _im=_im*rhs)
Return value: ***this**

```
float_complex& float_complex::operator/=(float rhs)
```

Divides ***this** by **rhs** and stores the result in ***this**.
(_re=_re/rhs, _im=_im/rhs)
Return value: ***this**

```
float_complex& float_complex::operator=(const float_complex& rhs)
```

Copies **rhs** to ***this**.
Return value: ***this**

float_complex& float_complex::operator+=(const float_complex& rhs)

Adds **rhs** to ***this** and stores the result in ***this**
Return value: ***this**

float_complex& float_complex::operator-=(const float_complex& rhs)

Subtracts **rhs** from ***this** and stores the result in ***this**.
Return value: ***this**

float_complex& float_complex::operator*=(const float_complex& rhs)

Multiplies ***this** by **rhs** and stores the result in ***this**.
Return value: ***this**

float_complex& float_complex::operator/=(const float_complex& rhs)

Divides ***this** by **rhs** and stores the result in ***this**.
Return value: ***this**

(b) float_complex Non-Member Function

Type	Definition Name	Description
Function	float_complex operator+(const float_complex& lhs)	Performs unary + operation of **lhs**.
	float_complex operator+(const float_complex& lhs, const float_complex& rhs)	Returns the result of adding **lhs** to **rhs**.
	float_complex operator+(const float_complex& lhs, const float& rhs)	
	float_complex operator+(const float& lhs, const float_complex& rhs)	
	float_complex operator-(const float_complex& lhs)	Performs unary - operation of **lhs**.
	float_complex operator-(const float_complex& lhs, const float_complex& rhs)	Returns the result of subtracting **rhs** from **lhs**.
	float_complex operator-(const float_complex& lhs, const float& rhs)	
	float_complex operator-(const float& lhs, const float_complex& rhs)	

Type	Definition Name	Description
Function	float_complex operator*(const float_complex& lhs, const float_complex& rhs)	Returns the result of multiplying **lhs** by **rhs**.
	float_complex operator*(const float_complex& lhs, const float& rhs)	
	float_complex operator*(const float& lhs, const float_complex& rhs)	
	float_complex operator/(const float_complex& lhs, const float_complex& rhs)	Returns the result of dividing **lhs** by **rhs**.
	float_complex operator/(const float_complex& lhs, const float& rhs)	
	float_complex operator/(const float& lhs, const float_complex& rhs)	Divides **lhs** by **rhs** and stores the quotient in **lhs**.
	bool operator==(const float_complex& lhs, const float_complex& rhs)	Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
	bool operator==(const float_complex& lhs, const float& rhs)	
	bool operator== (const float& lhs, const float_complex& rhs)	
	bool operator!=(const float_complex& lhs, const float_complex& rhs)	Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
	bool operator!=(const float_complex& lhs, const float& rhs)	
	bool operator!=(const float& lhs, const float_complex& rhs)	
	istream& operator>>(istream& is, float_complex& x)	Inputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part).
	ostream& operator<<(ostream& os, const float_complex& x)	Outputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part).
	float real(const float_complex& x)	Acquires the real part.
	float imag(const float_complex& x)	Acquires the imaginary part.
	float abs(const float_complex& x)	Calculates the absolute value.
	float arg(const float_complex& x)	Calculates the phase angle.
	float norm(const float_complex& x)	Calculates the absolute value of the square.

Type	Definition Name	Description
Function	float_complex conj(const float_complex& x)	Calculates the conjugate complex number.
	float_complex polar(const float& rho, const float& theta)	Calculates the **float_complex** value for a complex number with size **rho** and phase angle **theta**.
	float_complex cos(const float_complex& x)	Calculates the complex cosine.
	float_complex cosh(const float_complex& x)	Calculates the complex hyperbolic cosine.
	float_complex exp(const float_complex& x)	Calculates the exponent function.
	float_complex log(const float_complex& x)	Calculates the natural logarithm.
	float_complex log10(const float_complex& x)	Calculates the common logarithm.
	float_complex pow(const float_complex& x, int y)	Calculates **x** to the **y**th power.
	float_complex pow(const float_complex& x, const float& y)	
	float_complex pow(const float_complex& x, const float_complex& y)	
	float_complex pow(const float& x, const float_complex& y)	
	float_complex sin(const float_complex& x)	Calculates the complex sine.
	float_complex sinh(const float_complex& x)	Calculates the complex hyperbolic sine.
	float_complex sqrt(const float_complex& x)	Calculates the square root within the right half space.
	float_complex tan(const float_complex& x)	Calculates the complex tangent.
	float_complex tanh(const float_complex& x)	Calculates the complex hyperbolic tangent.

float_complex operator+(const float_complex& lhs)

Performs unary + operation of **lhs**.
Return value: **lhs**

float_complex operator+(const float_complex& lhs, const float_complex& rhs)

Returns the result of adding **lhs** to **rhs**.
Return value: **float_complex(lhs)**+=**rhs**

float_complex operator+(const float_complex& lhs, const float& rhs)

Returns the result of adding **lhs** to **rhs**.
Return value: **float_complex(lhs)**+=**rhs**

```
float_complex operator+(const float& lhs, const float_complex& rhs)
```

Returns the result of adding **lhs** to **rhs**.
Return value: **float_complex(lhs)**+=**rhs**

```
float_complex operator-(const float_complex& lhs)
```

Performs unary - operation of **lhs**.
Return value: **float_complex(-lhs.real(), -lhs.imag())**

```
float_complex operator-(const float_complex& lhs, const float_complex& rhs)
```

Returns the result of subtracting **rhs** from **lhs**.
Return value: **float_complex(lhs)**-=**rhs**

```
float_complex operator-(const float_complex& lhs, const float& rhs)
```

Returns the result of subtracting **rhs** from **lhs**.
Return value: **float_complex(lhs)**-=**rhs**

```
float_complex operator-(const float& lhs, const float_complex& rhs)
```

Returns the result of subtracting **rhs** from **lhs**.
Return value: **float_complex(lhs)**-=**rhs**

```
float_complex operator*(const float_complex& lhs, const float_complex& rhs)
```

Returns the result of multiplying **lhs** by **rhs**.
Return value: **float_complex(lhs)***=**rhs**

```
float_complex operator*(const float_complex& lhs, const float& rhs)
```

Returns the result of multiplying **lhs** by **rhs**.
Return value: **float_complex(lhs)***=**rhs**

```
float_complex operator*(const float& lhs, const float_complex& rhs)
```

Returns the result of multiplying **lhs** by **rhs**.
Return value: **float_complex(lhs)***=**rhs**

```
float_complex operator/(const float_complex& lhs, const float_complex& rhs)
```

Returns the result of dividing **lhs** by **rhs**.
Return value: **float_complex(lhs)**/=**rhs**

```
float_complex operator/(const float_complex& lhs, const float& rhs)
```

Returns the result of dividing **lhs** by **rhs**.
Return value: **float_complex(lhs)**/=**rhs**

```
float_complex operator/(const float& lhs, const float_complex& rhs)
```

Returns the result of dividing **lhs** by **rhs**.
Return value: **float_complex(lhs)**/=**rhs**

```
bool operator==(const float_complex& lhs, const float_complex& rhs)
```

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **float** type parameter, the imaginary part is assumed to be 0.0f.
Return value: **lhs.real()**==**rhs.real()** && **lhs.imag()**==**rhs.imag()**

```
bool operator==(const float_complex& lhs, const float& rhs)
```

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **float** type parameter, the imaginary part is assumed to be 0.0f.
Return value: **lhs.real()**==**rhs.real()** && **lhs.imag()**==**rhs.imag()**

```
bool operator==(const float& lhs, const float_complex& rhs)
```

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **float** type parameter, the imaginary part is assumed to be 0.0f.
Return value: **lhs.real()**==**rhs.real()** && **lhs.imag()**==**rhs.imag()**

```
bool operator!=(const float_complex& lhs, const float_complex& rhs)
```

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **float** type parameter, the imaginary part is assumed to be 0.0f.
Return value: **lhs.real()**!=**rhs.real()** || **lhs.imag()**!=**rhs.imag()**

```
bool operator!=(const float_complex& lhs, const float& rhs)
```

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **float** type parameter, the imaginary part is assumed to be 0.0f.
Return value: **lhs.real()**!=**rhs.real()** || **lhs.imag()**!=**rhs.imag()**

```
bool operator!=(const float& lhs, const float_complex& rhs)
```

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **float** type parameter, the imaginary part is assumed to be 0.0f.
Return value: **lhs.real()**!=**rhs.real()** || **lhs.imag()**!=**rhs.imag()**

```
istream& operator>>(istream& is, float_complex& x)
```

Inputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part).
The input value is converted to **float_complex**.
If **x** is input in a format other than the **u**, (**u**), or (**u**,**v**) format, **is.setstate(ios_base::failbit)** is called.
Return value: **is**

```
ostream& operator<<(ostream& os, const float_complex& x)
```

Outputs **x** to **os**.
The output format is **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part).
Return value: **os**

float real(const float_complex& x)

Acquires the real part.
Return value: **x.real()**

float imag(const float_complex& x)

Acquires the imaginary part.
Return value: **x.imag()**

float abs(const float_complex& x)

Calculates the absolute value.
Return value: (|**x.real()**|^2 + |**x.imag()**|^2)$^{1/2}$

float arg(const float_complex& x)

Calculates the phase angle.
Return value: **atan2f(x.imag(), x.real())**

float norm(const float_complex& x)

Calculates the absolute value of the square.
Return value: |**x.real()**|^2 + |**x.imag()**|^2

float_complex conj(const float_complex& x)

Calculates the conjugate complex number.
Return value: **float_complex(x.real(), (-1)*x.imag())**

float_complex polar(const float& rho, const float& theta)

Calculates the **float_complex** value for a complex number with size **rho** and phase angle (argument) **theta**.
Return value: **float_complex(rho*cosf(theta), rho*sinf(theta))**

float_complex cos(const float_complex& x)

Calculates the complex cosine.
Return value: **float_complex(cosf(x.real())*coshf(x.imag()),**
(-1)*sinf(x.real())*sinhf(x.imag()))

float_complex cosh(const float_complex& x)

Calculates the complex hyperbolic cosine.
Return value: **cos(float_complex((-1)*x.imag(), x.real()))**

float_complex& float_complex::operator-=(const float_complex& rhs)

Calculates the exponent function.
Return value: **expf(x.real())*cosf(x.imag()),expf(x.real())*sinf(x.imag())**

float_complex log(const float_complex& x)

Calculates the natural logarithm (base e).
Return value: **float_complex(logf(abs(x)), arg(x))**

float_complex log10(const float_complex& x)

Calculates the common logarithm (base 10).
Return value: **float_complex(log10f(abs(x)), arg(x)/logf(10))**

float_complex pow(const float_complex& x, int y)

Calculates **x** to the **y**th power.
If pow(0,0), a domain error will occur.
Return value: If **float_complex pow(const float_complex& x, const float_complex& y)**: exp(y*logf(x))
Otherwise: exp(y*log(x))

float_complex pow(const float_complex& x, const float& y)

Calculates **x** to the **y**th power.
If pow(0,0), a domain error will occur.
Return value: If **float_complex pow(const float_complex& x, const float_complex& y)**: exp(y*logf(x))
Otherwise: exp(y*log(x))

float_complex pow(const float_complex& x, const float_complex& y)

Calculates **x** to the **y**th power.
If pow(0,0), a domain error will occur.
Return value: If **float_complex pow(const float_complex& x, const float_complex& y)**: exp(y*logf(x))
Otherwise: exp(y*log(x))

float_complex pow(const float& x, const float_complex& y)

Calculates **x** to the **y**th power.
If pow(0,0), a domain error will occur.
Return value: If **float_complex pow(const float_complex& x, const float_complex& y)**: exp(y*logf(x))
Otherwise: exp(y*log(x))

float_complex sin(const float_complex& x)

Calculates the complex sine.
Return value: **float_complex(sinf(x.real())*coshf(x.imag()), cosf(x.real())*sinhf(x.imag()))**

float_complex sinh(const float_complex& x)

Calculates the complex hyperbolic sine.
Return value: **float_complex(0,-1)*sin(float_complex((-1)*x.imag(),x.real()))**

float_complex sqrt(const float_complex& x)

Calculates the square root within the right half space.
Return value: **float_complex(sqrtf(abs(x))*cosf(arg(x)/2), sqrtf(abs(x))*sinf(arg(x)/2))**

float_complex tan(const float_complex& x)

Calculates the complex tangent.
Return value: **sin(x)/cos(x)**

float_complex tanh(const float_complex& x)

Calculates the complex hyperbolic tangent.
Return value: **sinh(x)/cosh(x)**

(c) double_complex Class

Type	Definition Name	Description
Type	value_type	**double** type.
Variable	_re	Defines the real part of **double** precision.
	_im	Defines the imaginary part of **double** precision.
Function	double_complex(double re = 0.0, double im = 0.0)	Constructor.
	double_complex(const float_complex&)	
	double real() const	Acquires the real part.
	double imag() const	Acquires the imaginary part.
	double_complex& operator=(double rhs)	Copies **rhs** to the real part. 0.0 is assigned to the imaginary part.
	double_complex& operator+=(double rhs)	Adds **rhs** to the real part of ***this** and stores the sum in ***this**.
	double_complex& operator-=(double rhs)	Subtracts **rhs** from the real part of ***this** and stores the difference in ***this**.
	double_complex& operator*=(double rhs)	Multiplies ***this** by **rhs** and stores the product in ***this**.
	double_complex& operator/=(double rhs)	Divides ***this** by **rhs** and stores the quotient in ***this**.
	double_complex& operator=(const double_complex& rhs)	Copies **rhs**.
	double_complex& operator+=(const double_complex& rhs)	Adds **rhs** to ***this** and stores the sum in ***this**.
	double_complex& operator-=(const double_complex& rhs)	Subtracts **rhs** from ***this** and stores the difference in ***this**.
	double_complex& operator*=(const double_complex& rhs)	Multiplies ***this** by **rhs** and stores the product in ***this**.
	double_complex& operator/=(const double_complex& rhs)	Divides ***this** by **rhs** and stores the quotient in ***this**.

```
double_complex::double_complex(double re = 0.0, double im = 0.0)
```

Constructor of class **double_complex**.
The initial settings are as follows:
_re = re;
_im = im;

```
double_complex::double_complex(const float_complex&)
```

Constructor of class **double_complex**.
The initial settings are as follows:
_re = (double)rhs.real();
_im = (double)rhs.imag();

```
double double_complex::real() const
```

Acquires the real part.
Return value: **this->_re**

```
double double_complex::imag() const
```

Acquires the imaginary part.
Return value: **this->_im**

```
double_complex& double_complex::operator=(double rhs)
```

Copies **rhs** to the real part (**_re**).
0.0 is assigned to the imaginary part (**_im**).
Return value: ***this**

```
double_complex& double_complex::operator+=(double rhs)
```

Adds **rhs** to the real part (**_re**) and stores the result in the real part (**_re**).
The value of the imaginary part (**_im**) does not change.
Return value: ***this**

```
double_complex& double_complex::operator-= (double rhs)
```

Subtracts **rhs** from the real part (**_re**) and stores the result in the real part (**_re**).
The value of the imaginary part (**_im**) does not change.
Return value: ***this**

```
double_complex& double_complex::operator*=(double rhs)
```

Multiplies ***this** by **rhs** and stores the result in ***this**.
(_re=_re*rhs, _im=_im*rhs)
Return value: ***this**

```
double_complex& double_complex::operator/=(double rhs)
```

Divides ***this** by **rhs** and stores the result in ***this**.
(_re=_re/rhs, _im=_im/rhs)
Return value: ***this**

```
double_complex& double_complex::operator=(const double_complex& rhs)
```

Copies **rhs** to ***this**.
Return value: ***this**

```
double_complex& double_complex::operator+=(const double_complex& rhs)
```

Adds **rhs** to ***this** and stores the result in ***this**.
Return value: ***this**

```
double_complex& double_complex::operator-=(const double_complex& rhs)
```

Subtracts **rhs** from ***this** and stores the result in ***this**.
Return value: ***this**

```
double_complex& double_complex::operator*=(const double_complex& rhs)
```

Multiplies ***this** by **rhs** and stores the result in ***this**.
Return value: ***this**

```
double_complex& double_complex::operator/=(const double_complex& rhs)
```

Divides ***this** by **rhs** and stores the result in ***this**.
Return value: ***this**

(d) double_complex Non-Member Function

Type	Definition Name	Description
Function	double_complex operator+(const double_complex& lhs)	Performs unary + operation of **lhs**.
	double_complex operator+(const double_complex& lhs, const double_complex& rhs)	Returns the result of adding **rhs** to **lhs**.
	double_complex operator+(const double_complex& lhs, const double& rhs)	
	double_complex operator+(const double& lhs, const double_complex& rhs)	
	double_complex operator-(const double_complex& lhs)	Performs unary - operation of **lhs**.
	double_complex operator-(const double_complex& lhs, const double_complex& rhs)	Returns the result of subtracting **rhs** from **lhs**.
	double_complex operator-(const double_complex& lhs, const double& rhs)	
	double_complex operator-(const double& lhs, const double_complex& rhs)	

Type	Definition Name	Description
Function	double_complex operator*(const double_complex& lhs, const double_complex& rhs)	Returns the result of multiplying **lhs** by **rhs**.
	double_complex operator*(const double_complex& lhs, const double& rhs)	
	double_complex operator*(const double& lhs, const double_complex& rhs)	
	double_complex operator/(const double_complex& lhs, const double_complex& rhs)	Returns the result of dividing **lhs** by **rhs**.
	double_complex operator/(const double_complex& lhs, const double& rhs)	
	double_complex operator/(const double& lhs, const double_complex& rhs)	
	bool operator==(const double_complex& lhs, const double_complex& rhs)	Compares the real part of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
	bool operator==(const double_complex& lhs, const double& rhs)	
	bool operator==(const double& lhs, const double_complex& rhs)	
	bool operator!=(const double_complex& lhs, const double_complex& rhs)	Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
	bool operator!=(const double_complex& lhs, const double& rhs)	
	bool operator!=(const double& lhs, const double_complex& rhs)	
	istream& operator>>(istream& is, double_complex& x)	Inputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part).
	ostream& operator<<(ostream& os, const double_complex& x)	Outputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part).
	double real(const double_complex& x)	Acquires the real part.
	double imag(const double_complex& x)	Acquires the imaginary part.
	double abs(const double_complex& x)	Calculates the absolute value.
	double arg(const double_complex& x)	Calculates the phase angle.
	double norm(const double_complex& x)	Calculates the absolute value of the square.

Type	Definition Name	Description
Function	double_complex conj(const double_complex& x)	Calculates the conjugate complex number.
	double_complex polar(const double& rho, const double& theta)	Calculates the **double_complex** value for a complex number with size **rho** and phase angle **theta**.
	double_complex cos(const double_complex& x)	Calculates the complex cosine.
	double_complex cosh(const double_complex& x)	Calculates the complex hyperbolic cosine.
	double_complex exp(const double_complex& x)	Calculates the exponent function.
	double_complex log(const double_complex& x)	Calculates the natural logarithm.
	double_complex log10(const double_complex& x)	Calculates the common logarithm.
	double_complex pow(const double_complex& x, int y)	Calculates **x** to the **y**th power.
	double_complex pow(const double_complex& x, const double& y)	
	double_complex pow(const double_complex& x, const double_complex& y)	
	double_complex pow(const double& x, const double_complex& y)	
	double_complex sin(const double_complex& x)	Calculates the complex sine.
	double_complex sinh(const double_complex& x)	Calculates the complex hyperbolic sine.
	double_complex sqrt(const double_complex& x)	Calculates the square root within the right half space.
	double_complex tan(const double_complex& x)	Calculates the complex tangent.
	double_complex tanh(const double_complex& x)	Calculates the complex hyperbolic tangent.

```
double_complex operator+(const double_complex& lhs)
```

Performs unary + operation of **lhs**.
Return value: **lhs**

```
double_complex operator+(const double_complex& lhs, const double_complex& rhs)
```

Returns the result of adding **lhs** to **rhs**.
Return value: **double_complex(lhs)**+=**rhs**

```
double_complex operator+(const double_complex& lhs, const double& rhs)
```

Returns the result of adding **lhs** to **rhs**.
Return value: **double_complex(lhs)**+=**rhs**

```
double_complex operator+(const double& lhs, const double_complex& rhs)
```

Returns the result of adding **lhs** to **rhs**.
Return value: **double_complex(lhs)**+=**rhs**

```
double_complex operator-(const double_complex& lhs)
```

Performs unary - operation of **lhs**.
Return value: **double_complex(-lhs.real(), -lhs.imag())**

```
double_complex operator-(const double_complex& lhs, const double_complex& rhs)
```

Returns the result of subtracting **rhs** from **lhs**.
Return value: **double_complex(lhs)**-=**rhs**

```
double_complex operator-(const double_complex& lhs, const double& rhs)
```

Returns the result of subtracting **rhs** from **lhs**.
Return value: **double_complex(lhs)**-=**rhs**

```
double_complex operator-(const double& lhs, const double_complex& rhs)
```

Returns the result of subtracting **rhs** from **lhs**.
Return value: **double_complex(lhs)**-=**rhs**

```
double_complex operator*(const double_complex& lhs, const double_complex& rhs)
```

Returns the result of multiplying **lhs** by **rhs**.
Return value: **double_complex(lhs)***=**rhs**

```
double_complex operator*(const double_complex& lhs, const double& rhs)
```

Returns the result of multiplying **lhs** by **rhs**.
Return value: **double_complex(lhs)***=**rhs**

```
double_complex operator*(const double& lhs, const double_complex& rhs)
```

Returns the result of multiplying **lhs** by **rhs**.
Return value: **double_complex(lhs)***=**rhs**

```
double_complex operator/(const double_complex& lhs, const double_complex& rhs)
```

Returns the result of dividing **lhs** by **rhs**.
Return value: **double_complex(lhs)**/=**rhs**

double_complex operator/(const double_complex& lhs, const double& rhs)

Returns the result of dividing **lhs** by **rhs**.
Return value: **double_complex(lhs)**/=**rhs**

double_complex operator/(const double& lhs, const double_complex& rhs)

Returns the result of dividing **lhs** by **rhs**.
Return value: **double_complex(lhs)**/=**rhs**

bool operator==(const double_complex& lhs, const double_complex& rhs)

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **double** type parameter, the imaginary part is assumed to be 0.0.
Return value: lhs.real()==rhs.real() && lhs.imag()==rhs.imag()

bool operator==(const double_complex& lhs, const double& rhs)

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **double** type parameter, the imaginary part is assumed to be 0.0.
Return value: lhs.real()==rhs.real() && lhs.imag()==rhs.imag()

bool operator==(const double& lhs, const double_complex& rhs)

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **double** type parameter, the imaginary part is assumed to be 0.0.
Return value: lhs.real()==rhs.real() && lhs.imag()==rhs.imag()

bool operator!=(const double_complex& lhs, const double_complex& rhs)

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **double** type parameter, the imaginary part is assumed to be 0.0.
Return value: **lhs.real()**!=**rhs.real()** || **lhs.imag()**!=**rhs.imag()**

bool operator!=(const double_complex& lhs, const double& rhs)

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **double** type parameter, the imaginary part is assumed to be 0.0.
Return value: **lhs.real()**!=**rhs.real()** || **lhs.imag()**!=**rhs.imag()**

bool operator!=(const double& lhs, const double_complex& rhs)

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.
For a **double** type parameter, the imaginary part is assumed to be 0.0.
Return value: **lhs.real()**!=**rhs.real()** || **lhs.imag()**!=**rhs.imag()**

istream& operator>>(istream& is, double_complex& x)

Inputs complex number **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part).
The input value is converted to **double_complex**.
If **x** is input in a format other than the **u**, (**u**), or (**u**,**v**) format, **is.setstate(ios_base::failbit)** is called.
Return value: **is**

ostream& operator<<(ostream& os, const double_complex& x)

Outputs **x** to **os**.
The output format is **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part).
Return value: **os**

double real(const double_complex& x)

Acquires the real part.
Return value: **x.real()**

double imag(const double_complex& x)

Acquires the imaginary part.
Return value: **x.imag()**

double abs(const double_complex& x)

Calculates the absolute value.
Return value: $(|\mathbf{x.real()}|^2 + |\mathbf{x.imag()}|^2)^{1/2}$

double arg(const double_complex& x)

Calculates the phase angle.
Return value: **atan2(x.imag(), x.real())**

double norm(const double_complex& x)

Calculates the absolute value of the square.
Return value: $|\mathbf{x.real()}|^2 + |\mathbf{x.imag()}|^2$

double_complex conj(const double_complex& x)

Calculates the conjugate complex number.
Return value: **double_complex(x.real(), (-1)*x.imag())**

double_complex polar(const double& rho, const double& theta)

Calculates the **double_complex** value for a complex number with size **rho** and phase angle (argument) **theta**.
Return value: **double_complex(rho*cos(theta), rho*sin(theta))**

double_complex cos(const double_complex& x)

Calculates the complex cosine.
Return value: **double_complex(cos(x.real())*cosh(x.imag()),**
(-1)*sin(x.real())*sinh(x.imag()))

double_complex cosh(const double_complex& x)

Calculates the complex hyperbolic cosine.
Return value: **cos(double_complex((-1)*x.imag(), x.real()))**

double_complex exp(const double_complex& x)

Calculates the exponent function.
Return value: **exp(x.real())*cos(x.imag()),exp(x.real())*sin(x.imag())**

double_complex log(const double_complex& x)

Calculates the natural logarithm (base e).
Return value: **double_complex(log(abs(x)), arg(x))**

double_complex log10(const double_complex& x)

Calculates the common logarithm (base 10).
Return value: **double_complex(log10(abs(x)), arg(x)/log(10))**

double_complex pow(const double_complex& x, int y)

Calculates **x** to the **y**th power.
If pow(0,0), a domain error will occur.
Return value: exp(y*log(x))

double_complex pow(const double_complex& x, const double& y)

Calculates **x** to the **y**th power.
If pow(0,0), a domain error will occur.
Return value: exp(y*log(x))

double_complex pow(const double_complex& x, const double_complex& y)

Calculates **x** to the **y**th power.
If pow(0,0), a domain error will occur.
Return value: exp(y*log(x))

double_complex pow(const double& x, const double_complex& y)

Calculates **x** to the **y**th power.
If pow(0,0), a domain error will occur.
Return value: exp(y*log(x))

double_complex sin(const double_complex& x)

Calculates the complex sine
Return value: **double_complex(sin(x.real())*cosh(x.imag()), cos(x.real())*sinh(x.imag()))**

double_complex sinh(const double_complex& x)

Calculates the complex hyperbolic sine
Return value: **double_complex(0,-1)*sin(double_complex((-1)*x.imag(),x.real()))**

double_complex sqrt(const double_complex& x)

Calculates the square root within the right half space
Return value: **double_complex(sqrt(abs(x))*cos(arg(x)/2), sqrt(abs(x))*sin(arg(x)/2))**

```
double_complex tan(const double_complex& x)
```

Calculates the complex tangent.
Return value: **sin(x)/cos(x)**

```
double_complex tanh(const double_complex& x)
```

Calculates the complex hyperbolic tangent.
Return value: **sinh(x)/cosh(x)**

7.5.4 String Handling Class Library

The header file for the string handling class library is as follows:

- <string>
  Defines class **string**.

This class has no derivation.

(a) string Class

Type	Definition Name	Description
Type	iterator	**char*** type.
	const_iterator	**const char*** type.
Constant	npos	Maximum string length (**UNIT_MAX** characters).
Variable	s_ptr	Pointer to the memory area where the string is stored by the object.
	s_len	The length of the string stored by the object.
	s_res	Size of the allocated memory area to store string by the object.
Function	string(void)	Constructor.
	string(const string& str, size_t pos = 0, size_t n = npos)	
	string(const char* str, size_t n)	
	string(const char* str)	
	string(size_t n, char c)	
	~string()	Destructor.
	string& operator=(const string& str)	Assigns **str**.
	string& operator=(const char* str)	
	string& operator=(char c)	Assigns **c**.
	iterator begin()	Calculates the start pointer of the string.
	const_iterator begin() const	
	iterator end()	Calculates the end pointer of the string.
	const_iterator end() const	
	size_t size() const	Calculates the length of the stored string.
	size_t length() const	
	size_t max_size() const	Calculates the size of the allocated memory area.
	void resize(size_t n, char c)	Changes the storable string length to **n**.
	void resize(size_t n)	Changes the storable string length to **n**.
	size_t capacity() const	Calculates the size of the allocated memory area.
	void reserve(size_t res_arg = 0)	Performs re-allocation of the memory area.
	void clear()	Clears the stored string.
	bool empty() const	Checks whether the stored string length is 0.

Type	Definition Name	Description
Function	const char& operator[](size_t pos) const	References **s_ptr[pos]**.
	char& operator[](size_t pos)	
	const char& at(size_t pos) const	
	char& at(size_t pos)	
	tring& operator+=(const string& str)	Adds string **str**.
	string& operator+=(const char* str)	
	string& operator+=(char c)	Adds character **c**.
	string& append(const string& str)	Adds string **str**.
	string& append(const char* str)	
	string& append(const string& str, size_t pos, size_t n)	Adds **n** characters of string **str** at object position **pos**.
	string& append(const char* str, size_t n)	Adds **n** characters to string **str**.
	string& append(size_t n, char c)	Adds **n** characters, each of which is **c**.
	string& assign(const string& str)	Assigns string **str**.
	string& assign(const char* str)	
	string& assign(const string& str, size_t pos, size_t n)	Add **n** characters to string **str** at position **pos**.
	string& assign(const char* str, size_t n)	Assigns **n** characters of string **str**.
	string& assign(size_t n, char c)	Assigns **n** characters, each of which is **c**.
	string& insert(size_t pos1, const string& str)	Inserts string **str** to position **pos1**.
	string& insert(size_t pos1, const string& str, size_t pos2, size_t n)	Inserts **n** characters starting from position **pos2** of string **str** to position **pos1**.
	string& insert(size_t pos, const char* str, size_t n)	Inserts **n** characters of string **str** to position **pos**.
	string& insert(size_t pos, const char* str)	Inserts string **str** to position **pos**.
	string& insert(size_t pos, size_t n, char c)	Inserts a string of **n** characters, each of which is **c**, to position **pos**.
	iterator insert(iterator p, char c = char())	Inserts character **c** before the string specified by **p**.
	void insert(iterator p, size_t n, char c)	Inserts **n** characters, each of which is **c**, before the character specified by **p**.
	string& erase(size_t pos = 0, size_t n = npos)	Deletes **n** characters from position **pos**.

Type	Definition Name	Description
Function	iterator erase(iterator position)	Deletes the character referenced by **position**.
	iterator erase(iterator first, iterator last)	Deletes the characters in range [**first, last**].
	string& replace(size_t pos1, size_t n1, const string& str)	Replaces the string of **n1** characters starting from position **pos1** with string **str**.
	string& replace(size_t pos1, size_t n1, const char* str)	
	string& replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)	Replaces the string of **n1** characters starting from position **pos1** with string of **n2** characters from position **pos2** of **str**.
	string& replace(size_t pos, size_t n1, const char* str, size_t n2)	Replaces the string of **n1** characters starting from position **pos** with string **str** of **n2** characters.
	string& replace(size_t pos, size_t n1, size_t n2, char c)	Replaces the string of **n1** characters starting from position **pos** with **n2** characters, each of which is **c**.
	string& replace(iterator i1, iterator i2, const string& str)	Replaces the string from position **i1** to **i2** with string **str**.
	string& replace(iterator i1, iterator i2, const char* str)	
	string& replace(iterator i1, iterator i2, const char* str, size_t n)	Replaces the string from position **i1** to **i2** with **n** characters of string **str**.
	string& replace(iterator i1, iterator i2, size_t n, char c)	Replaces the string from position **i1** to **i2** with **n** characters, each of which is **c**.
	size_t copy(char* str, size_t n, size_t pos = 0) const	Copies the first **n** characters of string **str** to position **pos**.
	void swap(string& str)	Swaps ***this** with string **str**.

Type	Definition Name	Description
Function	const char* c_str() const	References the pointer to the memory area where the string is stored.
	const char* data() const	
	size_t find(const string& str, size_t pos = 0) const	Finds the position where the string same as string **str** first appears after position **pos**.
	size_t find(const char* str, size_t pos = 0) const	
	size_t find(const char* str, size_t pos, size_t n) const	Finds the position where the string same as **n** characters of **str** first appears after position **pos**.
	size_t find(char c, size_t pos = 0) const	Finds the position where character **c** first appears after position **pos**.
	size_t rfind(const string& str, size_t pos = npos) const	Finds the position where a string same as string **str** appears most recently before position **pos**.
	size_t rfind(const char* str, size_t pos = npos) const	
	size_t rfind(const char* str, size_t pos, size_t n) const	Finds the position where the string same as **n** characters of **str** appears most recently before position **pos**.
	size_t rfind(char c, size_t pos = npos) const	Finds the position where character **c** appears most recently before position **pos**.
	size_t find_first_of(const string& str, size_t pos = 0) const	Finds the position where any character included in string **str** first appears after position **pos**.
	size_t find_first_of(const char* str, size_t pos = 0) const	
	size_t find_first_of(const char* str, size_t pos, size_t n) const	Finds the position where any character included in **n** characters of string **str** first appears after position **pos**.
	size_t find_first_of(char c, size_t pos = 0) const	Finds the position where character **c** first appears after position **pos**.
	size_t find_last_of(const string& str, size_t pos = npos) const	Finds the position where any character included in string **str** appears most recently before position **pos**.
	size_t find_last_of(const char* str, size_t pos = npos) const	
	size_t find_last_of(const char* str, size_t pos, size_t n) const	Finds the position where any character included in **n** characters of string **str** appears most recently before position **pos**.

Type	Definition Name	Description
Function	size_t find_last_of(char c, size_t pos = npos) const	Finds the position where character **c** appears most recently before position **pos**.
	size_t find_first_not_of(const string& str, size_t pos = 0) const	Finds the position where a character different from any character included in string **str** first appears after position **pos**
	size_t find_first_not_of(const char* str, size_t pos = 0) const	
	size_t find_first_not_of(const char* str, size_t pos, size_t n) const	Finds the position where a character different from any character in the first **n** characters of string **str** appears after position **pos**.
	size_t find_first_not_of(char c, size_t pos = 0) const	Finds the position where a character different from **c** first appears after position **pos**.
	size_t find_last_not_of(const string& str, size_t pos = npos) const	Finds the position where a character different from any character included in string **str** appears most recently before position **pos**.
	size_t find_last_not_of(const char* str, size_t pos = npos) const	
	size_t find_last_not_of(const char* str, size_t pos, size_t n) const	Finds the position where a character different from any character in the first **n** characters of string **str** appears most recently before position **pos**.
	size_t find_last_not_of(char c, size_t pos = npos) const	Finds the position where a character different from **c** appears most recently before position **pos**.
	string substr(size_t pos = 0, size_t n = npos) const	Creates an object from a string in the range [**pos**,**n**] of the stored string.
	int compare(const string& str) const	Compares the string with string **str**.
	int compare(size_t pos1, size_t n1, const string& str) const	Compares **n1** characters from position **pos1** of ***this** with **str**.
	int compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const	Compares the string of **n1** characters from position **pos1** with the string of **n2** characters from position **pos2** of string **str**.
	int compare(const char* str) const	Compares ***this** with string **str**.
	int compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const	Compares the string of **n1** characters from position **pos1** with **n2** characters of string **str**.

string::string(void)

Sets as follows:
s_ptr = 0;
s_len = 0;
s_res = 1;

string::string(const string& str, size_t pos = 0, size_t n = npos)

Copies **str**. Note that **s_len** will be the smaller value of **n** and **s_len**.

string::string(const char* str, size_t n)

Sets as follows:
s_ptr = **str**;
s_len = **n**;
s_res = **n** + 1;

string::string(const char* str)

Sets as follows:
s_ptr = **str**;
s_len = length of string **str**;
s_res = length of string **str** + 1;

string::string(size_t n, char c)

Sets as follows:
s_ptr = string of **n** characters, each of which is **c**
s_len = **n**;
s_res = **n** + 1;

string::~string()

Destructor of class **string**.
Deallocates the memory area where the string is stored.

string& string::operator(const string& str)

Assigns the data of **str**.
Return value: ***this**

string& string::operator=(const char* str)

Creates a **string** object from **str** and assigns its data to the **string** object.
Return value: ***this**

string& string::operator=(char c)

Creates a **string** object from **c** and assigns its data to the **string** object.
Return value: ***this**

```
string::iterator string::begin()
```

Calculates the start pointer of the string.
Return value: Start pointer of the string.

```
string::const_iterator string::begin() const
```

Calculates the start pointer of the string.
Return value: Start pointer of the string.

```
string::iterator string::end()
```

Calculates the end pointer of the string.
Return value: End pointer of the string.

```
string::const_iterator string::end() const
```

Calculates the end pointer of the string.
Return value: End pointer of the string.

```
size_t string::size() const
```

Calculates the length of the stored string.
Return value: Length of the stored string.

```
size_t string::length() const
```

Calculates the length of the stored string.
Return value: Length of the stored string.

```
size_t string::max_size() const
```

Calculates the size of the allocated memory area.
Return value: Size of the allocated area.

```
void string::resize(size_t n, char c)
```

Changes the number of characters in the string that can be stored by the object to **n**.
If **n**<=**size()**, replaces the string with the original string with length **n**.
If **n**>**size()**, replaces the string with a string that has **c** appended to the end so that the length will be equal to **n**.
The length must be **n**<=**max_size()**.
If **n**>**max_size()**, the string length is **n**=**max_size()**.

```
void string::resize(size_t n)
```

Changes the number of characters in the string that can be stored by the object to **n**.
If **n**<=**size()**, replaces the string with the original string with length **n**.
The length must be **n**<=**max_size**.

```
size_t string::capacity() const
```

Calculates the size of the allocated memory area.
Return value: Size of the allocated memory area

void string::reserve(size_t res_arg = 0)

Re-allocates the memory area.
After **reserve()**, **capacity()** will be equal to or larger than the **reserve()** parameter.
When the memory area is re-allocated, all references, pointers, and **iterator** that references the elements of the numeric sequence become invalid.

void string::clear()

Clears the stored string.

bool string::empty() const

Checks whether the number of characters in the stored string is 0.
Return value: If the length of the stored string is 0: **true**
If the length of the stored string is not zero: **false**

const char& string::operator[ ](size_t pos) const

References **s_ptr[pos]**.
Return value: If **n**< **s_len**: **s_ptr [pos]**
If **n**>= **s_len**: '\0'

char& string::operator[ ](size_t pos)

References **s_ptr[pos]**.
Return value: If **n**< **s_len**: **s_ptr [pos]**
If **n**>= **s_len**: '\0'

const char& string::at(size_t pos) const

References **s_ptr[pos]**.
Return value: If **n**< **s_len**: **s_ptr [pos]**
If **n**>= **s_len**: '\0'

char& string::at(size_t pos)

References **s_ptr[pos]**.
Return value: If **n**< **s_len**: **s_ptr [pos]**
If **n**>= **s_len**: '\0'

string& string::operator+=(const string& str)

Appends the string stored in **str** to the object.
Return value: ***this**

string& string::operator+=(const char* str)

Creates a **string** object from **str** and adds the string to the object.
Return value: ***this**

```
string& string::operator+=(char c)
```

Creates a **string** object from **c** and adds the string to the object.
Return value: ***this**

```
string& string::append(const string& str)
```

Appends string **str** to the object.
Return value: ***this**

```
string& string::append(const char* str)
```

Appends string **str** to the object.
Return value: ***this**

```
string& string::append(const string& str, size_t pos, size_t n);
```

Appends **n** characters of string **str** to the object position **pos**.
Return value: ***this**

```
string& string::append(const char* str, size_t n)
```

Appends **n** characters of string **str** to the object.
Return value: ***this**

```
string& string::append(size_t n, char c)
```

Appends **n** characters, each of which is **c**, to the object.
Return value: ***this**

```
string& string::assign(const string& str)
```

Assigns string **str**.
Return value: ***this**

```
string& string::assign(const char* str)
```

Assigns string **str**.
Return value: ***this**

```
string& string::assign(const string& str, size_t pos, size_t n)
```

Assigns **n** characters of string **str** to position **pos**.
Return value: ***this**

```
string& string::assign(const char* str, size_t n)
```

Assigns **n** characters of string **str**.
Return value: ***this**

```
string& string::assign(size_t n, char c)
```

Assigns **n** characters, each of which is **c**.
Return value: ***this**

```
string& string::insert(size_t pos1, const string& str)
```

Inserts string **str** to position **pos1**.
Return value: ***this**

```
string& string::insert(size_t pos1, const string& str, size_t pos2, size_t n)
```

Inserts **n** characters starting from position **pos2** of string **str** to position **pos1**.
Return value: ***this**

```
string& string::insert(size_t pos, const char* str, size_t n)
```

Inserts **n** characters of string **str** to position **pos**.
Return value: ***this**

```
string& string::insert(size_t pos, const char* str)
```

Inserts string **str** to position **pos**.
Return value: ***this**

```
string& string::insert(size_t pos, size_t n, char c)
```

Inserts a string of **n** characters, each of which is **c**, to position **pos**.
Return value: ***this**

```
string::iterator string::insert(iterator p, char c = char())
```

Inserts character **c** before the string specified by **p**.
Return value: The inserted character

```
void string::insert(iterator p, size_t n, char c)
```

Inserts **n** characters, each of which is **c**, before the character specified by **p**.

```
string& string::erase(size_t pos = 0, size_t n = npos)
```

Deletes **n** characters starting from position **pos**.
Return value: ***this**

```
iterator string::erase(iterator position)
```

Deletes the character referenced by **position**.
Return value: If the next **iterator** of the element to be deleted exists: The next **iterator** of the deleted element
If the next **iterator** of the element to be deleted does not exist: **end()**

```
iterator string::erase(iterator first, iterator last)
```

Deletes the characters in range [**first**, **last**].
Return value: If the next **iterator** of **last** exists: The next **iterator** of **last**
If the next **iterator** of **last** does not exist: **end()**

```
string& string::replace(size_t pos1, size_t n1, const string& str)
```

Replaces the string of **n1** characters starting from position **pos1** with string **str**.
Return value: ***this**

```
string& string::replace(size_t pos1, size_t n1, const char* str)
```

Replaces the string of **n1** characters starting from position **pos1** with string **str**.
Return value: ***this**

```
string& string::replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)
```

Replaces the string of **n1** characters starting from position **pos1** with the string of **n2** characters starting from position **pos2** in string **str**.
Return value: ***this**

```
string& string::replace(size_t pos, size_t n1, const char* str, size_t n2)
```

Replaces the string of **n1** characters starting from position **pos1** with **n2** characters of string **str**.
Return value: ***this**

```
string& string::replace(size_t pos, size_t n1, size_t n2, char c)
```

Replaces the string of **n1** characters starting from position **pos** with **n2** characters, each of which is **c**.
Return value: ***this**

```
string& string::replace(iterator i1, iterator i2, const string& str)
```

Replaces the string from position **i1** to **i2** with string **str**.
Return value: ***this**

```
string& string::replace(iterator i1, iterator i2, const char* str)
```

Replaces the string from position **i1** to **i2** with string **str**.
Return value: ***this**

```
string& string::replace(iterator i1, iterator i2, const char* str, size_t n)
```

Replaces the string from position **i1** to **i2** with **n** characters of string **str**
Return value: ***this**

```
string& string::replace(iterator i1, iterator i2, size_t n, char c)
```

Replaces the string from position **i1** to **i2** with **n** characters, each of which is **c**.
Return value: ***this**

```
size_t string::copy(char* str, size_t n, size_t pos = 0) const
```

Copies **n** characters of string **str** to position **pos**.
Return value: **rlen**

```
void string::swap(string& str)
```

Swaps ***this** with string **str**.

```
const char* string::c_str() const
```

References the pointer to the memory area where the string is stored.
Return value: **s_ptr**

```
const char* string::data() const
```

References the pointer to the memory area where the string is stored.
Return value: **s_ptr**

```
size_t string::find(const string& str, size_t pos = 0) const
```

Finds the position where the string same as string **str** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find (const char* str, size_t pos = 0) const
```

Finds the position where the string same as string **str** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find(const char* str, size_t pos, size_t n) const
```

Finds the position where the string same as **n** characters of string **str** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find(char c, size_t pos = 0) const
```

Finds the position where character **c** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::rfind(const string& str, size_t pos = npos) const
```

Finds the position where a string same as string **str** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::rfind(const char* str, size_t pos = npos) const
```

Finds the position where a string same as string **str** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::rfind(const char* str, size_t pos, size_t n) const
```

Finds the position where the string same as **n** characters of string **str** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::rfind(char c, size_t pos = npos) const
```

Finds the position where character **c** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::find_first_of(const string& str, size_t pos = 0) const
```

Finds the position where any character included in string **str** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find_first_of(const char* str, size_t pos = 0) const
```

Finds the position where any character included in string **str** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find_first_of(const char* str, size_t pos, size_t n) const
```

Finds the position where any character included in **n** characters of string **str** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find_first_of(char c, size_t pos = 0) const
```

Finds the position where character **c** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find_last_of(const string& str, size_t pos = npos) const
```

Finds the position where any character included in string **str** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::find_last_of(const char* str, size_t pos = npos) const
```

Finds the position where any character included in string **str** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::find_last_of(const char* str, size_t pos, size_t n) const
```

Finds the position where any character included in **n** characters of string **str** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::find_last_of(char c, size_t pos = npos) const
```

Finds the position where character **c** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::find_first_not_of(const string& str, size_t pos = 0) const
```

Finds the position where a character different from any character included in string **str** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find_first_not_of(const char* str, size_t pos = 0) const
```

Finds the position where a character different from any character included in string **str** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find_first_not_of(const char* str, size_t pos, size_t n) const
```

Finds the position where a character different from any character in the first **n** characters of string **str** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find_first_not_of(char c, size_t pos = 0) const
```

Finds the position where a character different from character **c** first appears after position **pos**.
Return value: Offset of string.

```
size_t string::find_last_not_of(const string& str, size_t pos = npos) const
```

Finds the position where a character different from any character included in string **str** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::find_last_not_of(const char* str, size_t pos = npos) const
```

Finds the position where a character different from any character included in string **str** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::find_last_not_of(const char* str, size_t pos, size_t n) const
```

Finds the position where a character different from any character in the first **n** characters of string **str** appears most recently before position **pos**.
Return value: Offset of string.

```
size_t string::find_last_not_of(char c, size_t pos = npos) const
```

Finds the position where a character different from character **c** appears most recently before position **pos**.
Return value: Offset of string.

```
string string::substr(size_t pos = 0, size_t n = npos) const
```

Creates an object from a string in the range [**pos**,**n**] of the stored string.
Return value: Object with a string in the range [**pos**,**n**].

int string::compare(const string& str) const

Compares the string with string **str**.
Return value: If the strings are the same: 0
If the strings are different: 1 when **this->s_len** > **str.s_len**,
-1 when **this->s_len** < **str.s_len**

int string::compare(size_t pos1, size_t n1, const string& str) const

Compares a string of **n1** characters starting from position **pos1** of *this with string **str**.
Return value: If the strings are the same: 0
If the strings are different: 1 when **this->s_len** > **str.s_len**,
-1 when **this->s_len** < **str.s_len**

int string::compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const

Compares a string of **n1** characters starting from position **pos1** with the string of **n2** characters from position **pos2** of string **str**.
Return value: If the strings are the same: 0
If the strings are different: 1 when **this->s_len** > **str.s_len**,
-1 when **this->s_len** < **str.s_len**

int string::compare(const char* str) const

Compares *this with string **str**.
Return value: If the strings are the same: 0
If the strings are different: 1 when **this->s_len** > **str.s_len**,
-1 when **this->s_len** < **str.s_len**

int string::compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const

Compares the string of **n1** characters from position **pos1** with **n2** characters of string **str**.
Return value: If the strings are the same: 0
If the strings are different: 1 when **this->s_len** > **str.s_len**,
-1 when **this->s_len** < **str.s_len**

(b) string Class Manipulators

Type	Definition Name	Description
Function	string operator +(const string& lhs, const string& rhs)	Appends the string (or characters) of **rhs** to the string (or characters) of **lhs**, creates an object and stores the string in the object.
	string operator+(const char* lhs, const string& rhs)	
	string operator+(char lhs, const string& rhs)	
	string operator+(const string& lhs, const char* rhs)	
	string operator+(const string& lhs, char rhs)	
	bool operator==(const string& lhs, const string& rhs)	Compares the string of **lhs** with the string of **rhs**.
	bool operator==(const char* lhs, const string& rhs)	
	bool operator==(const string& lhs, const char* rhs)	

Type	Definition Name	Description
Function	bool operator!=(const string& lhs, const string& rhs)	Compares the string of **lhs** with the string of **rhs**.
	bool operator!=(const char* lhs, const string& rhs)	
	bool operator!=(const string& lhs, const char* rhs)	
	bool operator<(const string& lhs, const string& rhs)	Compares the string length of **lhs** with the string length of **rhs**.
	bool operator<(const char* lhs, const string& rhs)	Compares the string length of **lhs** with the string length of **rhs**.
	bool operator<(const string& lhs, const char* rhs)	
	bool operator>(const string& lhs, const string& rhs)	Compares the string length of **lhs** with the string length of **rhs**.
	bool operator>(const char* lhs, const string& rhs)	
	bool operator>(const string& lhs, const char* rhs)	
	bool operator<=(const string& lhs, const string& rhs)	Compares the string length of **lhs** with the string length of **rhs**.
	bool operator<=(const char* lhs, const string& rhs)	
	bool operator<=(const string& lhs, const char* rhs)	
	bool operator>=(const string& lhs, const string& rhs)	Compares the string length of **lhs** with the string length of **rhs**.
	bool operator>=(const char* lhs, const string& rhs)	
	bool operator>=(const string& lhs, const char* rhs)	
	void swap(string& lhs, string& rhs)	Swaps the string of **lhs** with the string of **rhs**.
	istream& operator>>(istream& is, string& str)	Extracts the string to **str**.
	ostream& operator<<(ostream& os, const string& str)	Inserts string **str**.
	istream& getline(istream& is, string& str, char delim)	Extracts a string from **is** and appends it to **str**. If **delim** is found in the string, input is stopped.
	istream& getline(istream& is, string& str)	Extracts a string from **is** and appends it to **str**. If a new-line character is detected, input is stopped.

```
string operator+(const string& lhs, const string& rhs)
```

Appends the string (characters) of **lhs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.
Return value: Object where the linked strings are stored.

```
string operator+(const char* lhs, const string& rhs)
```

Appends the string (characters) of **lhs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.
Return value: Object where the linked strings are stored.

string operator+(char lhs, const string& rhs)

Appends the string (characters) of **lhs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.
Return value: Object where the linked strings are stored.

string operator+(const string& lhs, const char* rhs)

Appends the string (characters) of **lhs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.
Return value: Object where the linked strings are stored.

string operator+(const string& lhs, char rhs)

Appends the string (characters) of **lhs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.
Return value: Object where the linked strings are stored.

bool operator==(const string& lhs, const string& rhs)

Compares the string of **lhs** with the string of **rhs**.
Return value: If the strings are the same: **true**
If the strings are different: **false**

bool operator==(const char* lhs, const string& rhs)

Compares the string of **lhs** with the string of **rhs**.
Return value: If the strings are the same: **true**
If the strings are different: **false**

bool operator==(const string& lhs, const char* rhs)

Compares the string of **lhs** with the string of **rhs**.
Return value: If the strings are the same: **true**
If the strings are different: **false**

bool operator!=(const string& lhs, const string& rhs)

Compares the string of **lhs** with the string of **rhs**.
Return value: If the strings are the same: **false**

bool operator!=(const char* lhs, const string& rhs)

Compares the string of **lhs** with the string of **rhs**.
Return value: If the strings are the same: **false**

bool operator!=(const string& lhs, const char* rhs)

Compares the string of **lhs** with the string of **rhs**.
Return value: If the strings are the same: **false**
If the strings are different: **true**

bool operator<(const string& lhs, const string& rhs)

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** < **rhs.s_len**: **true**
If **lhs.s_len** >= **rhs.s_len**: **false**

bool operator<(const char* lhs, const string& rhs)

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** < **rhs.s_len**: **true**
If **lhs.s_len** >= **rhs.s_len**: **false**

bool operator<(const string& lhs, const char* rhs)

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** < **rhs.s_len**: **true**
If **lhs.s_len** >= **rhs.s_len**: **false**

bool operator>(const string& lhs, const string& rhs)

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** > **rhs.s_len**: **true**

bool operator>(const char* lhs, const string& rhs)

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** > **rhs.s_len**: **true**

bool operator>(const string& lhs, const char* rhs)

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** > **rhs.s_len**: **true**
If **lhs.s_len** <= **rhs.s_len**: **false**

bool operator<=(const string& lhs, const string& rhs)

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** <= **rhs.s_len**: **true**
If **lhs.s_len** > **rhs.s_len**: **false**

bool operator<=(const char* lhs, const string& rhs)

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** <= **rhs.s_len**: **true**
If **lhs.s_len** > **rhs.s_len**: **false**

bool operator<=(const string& lhs, const char* rhs)

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** <= **rhs.s_len**: **true**
If **lhs.s_len** > **rhs.s_len**: **false**

```
bool operator>=(const string& lhs, const string& rhs)
```

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** >= **rhs.s_len**: **true**
If **lhs.s_len** < **rhs.s_len**: **false**

```
bool operator>=(const char* lhs, const string& rhs)
```

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** >= **rhs.s_len**: **true**
If **lhs.s_len** < **rhs.s_len**: **false**

```
bool operator>=(const string& lhs, const char* rhs)
```

Compares the string length of **lhs** with the string length of **rhs**.
Return value: If **lhs.s_len** >= **rhs.s_len**: **true**
If **lhs.s_len** < **rhs.s_len**: **false**

```
void swap(string& lhs,string& rhs)
```

Swaps the string of **lhs** with the string of **rhs**.

```
istream& operator>>(istream& is, string& str)
```

Extracts a string to **str**.
Return value: **is**

```
ostream& operator<<(ostream& os, const string& str)
```

Inserts string **str**.
Return value: **os**

```
istream& getline(istream& is, string& str, char delim)
```

Extracts a string from **is** and appends it to **str**.
If **delim** is found in the string, the input is stopped.
Return value: **is**

```
istream& getline(istream& is, string& str)
```

Extracts a string from **is** and appends it to **str**.
If a new-line character is found, the input is stopped.
Return value: **is**

7.6 Unsupported Libraries

Table 7.16 lists the libraries which are specified in the C language specifications but not supported by this compiler.

Table 7.16 Unsupported Libraries

No.	Header File	Library Names
1	locale.h*1	setlocale, localeconv
2	signal.h*1	signal, raise
3	stdio.h	remove, rename, tmpfile, tmpnam, fgetpos, fsetpos
4	stdlib.h	abort, atexit, exit, _Exit, getenv, system, mblen, mbtowc, wctomb, mbstowcs, wcstombs
5	string.h	strcoll, strxfrm
6	time.h	clock, difftime, mktime, time, asctime, ctime, gmtime, localtime, strftime
7	wctype.h	iswalnum, iswalpha, iswblank, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, iswctype, wctype, towlower, towupper, towctrans, wctrans
8	wchar.h	wcsftime, wcscoll, wcsxfrm, wctob, mbrtowc, wcrtomb, mbsrtowcs, wcsrtombs

Note The header file is not supported.

8. STARTUP

This chapter describes the startup routine.

8.1 Overview

Running a program written in C/C++ requires a separate program to create ROM images for inclusion in the target system and to start the user program (main function). This program is called the startup routine.
The user must create the startup routines for user programs. Renesas provides the source code for a startup routine with its integrated development environment (IDE) for the RX Family so that the user has a startup routine that can be adjusted to suit the target system.

8.2 File Contents

Startup routine that The Renesas integrated development environment (IDE) for RX Family supplies is as follows:

Table 8.1 List of Programs Created in Integrated Development Environment

	File Name	Description
(a)	resetprg.c	Initial setting routine (reset vector function)
(b)	intprg.c	Vector function definitions
(c)	vecttbl.c	Fixed vector table[*1]
(d)	dbsct.c	Section initialization processing (table)
(e)	lowsrc.c	Low-level interface routine (C language part)
(f)	lowlvl.src	Low-level interface routine (assembly language part)
(g)	sbrk.c	Low-level interface routine (**sbrk** function)
(h)	typedefine.h	Type definition header
(i)	vect.h	Vector function header
(j)	stacksct.h	Stack size settings
(k)	lowsrc.h	Low-level interface routine (C language header)
(l)	sbrk.h	Low-level interface routine (**sbrk** function header)

Notes 1. This is for the RXv1 instruction-set architecture.
For an RX instruction-set architecture other than the RXv1 instruction-set architecture, this becomes the "exception vector table".

8.3 Startup Program Creation

Here, processing to prepare the environment for program execution is described. However, the environment for program execution will differ among user systems, and so a program to set the execution environment must be created according to the specifications of the user system.
This section describes the standard startup program. The startup program for an application that uses the PIC/PID function needs special processing; refer also to section 8.5.7 Application Startup.
A summary of the necessary procedures is given below.

- Fixed vector table setting
  Sets the fixed vector table to initiate the initial setting routine (**PowerOn_Reset_PC**) at a power-on reset. In addition to the reset vector, processing routines, such as, privileged instruction exception, access exception, undefined instruction exception, floating-point exception, and nonmaskable interrupt, can be registered to the fixed vector table.
- Initial setting
  Performs the procedures required to reach the **main** function. Registers and sections are initialized and various initial setting routines are called.

- Low-level interface routine creation
  Routines providing an interface between the user system and library functions which are necessary when standard I/O (**stdio.h**, **ios**, **streambuf**, **istream**, and **ostream**) and memory management libraries (**stdlib.h** and **new**) are used.
- Termination processing routine (exit, atexit, and abort)* creation
  Processing for terminating the program is performed.

Note * When using the C library function **exit**, **atexit**, or **abort** to terminate a program, these functions must be created as appropriate to the user system.
When using the C++ program or C library macro **assert**, the **abort** function must always be created.

8.3.1 Fixed Vector Table Setting

To call the initial setting routine (**PowerOn_Reset_PC**) at a power-on reset, set the address of **PowerOn_Reset_PC** to the reset vector of the fixed vector table. A coding example is shown below.

In addition to the reset vector, processing routines, such as, privileged instruction exception, access exception, undefined instruction exception, floating-point exception, and nonmaskable interrupt, can be registered to the fixed vector table.

For details on the fixed vector table, refer to the hardware manual.

Example:

```
extern void PowerOn_Reset_PC(void);

#pragma section C FIXEDVECT    /* Outputs RESET_Vectors to the FIXEDVECT   */
                               /* section by #pragma section declaration.  */
                               /* Allocates the FIXEDVECT section to reset */
                               /* vector by the start option at linkage.   */
void (*const RESET_Vectors[])(void)={
    PowerOn_Reset_PC,
};
```

8.3.2 Initial Setting

The initial setting routine (**PowerOn_Reset_PC**) is a function that contains the procedures required before and after executing the **main** function. Processings required in the initial setting routine are described below in order.

(1) Initialization of PSW for Initial Setting Processing
The PSW register necessary for performing the initial setting processing is initialized. For example, disabling interrupts is set in PSW during the initial setting processing to prevent from accepting interrupts.
All bits in PSW are initialized to 0 at a reset, and the interrupt enable bit (I bit) is also initialized to 0 (interrupt disabled state).

(2) Initialization of Stack Pointer
The stack pointer (USP register and ISP register) is initialized. The **#pragma entry** declaration for the **PowerOn_Reset_PC** function makes the compiler automatically create the ISP/USP initialization code at the beginning of the function.
This procedure does not have to be written because the **PowerOn_Reset_PC** function is declared by **#pragma entry**.

(3) Initialization of General Registers Used as Base Registers
When the **base** option is used in the compiler, general registers used as base addresses in the entire program need to be initialized. The **#pragma entry** declaration for the **PowerOn_Reset_PC** function makes the compiler automatically create the initialization code for each register at the beginning of the function.
This procedure does not have to be written because the **PowerOn_Reset_PC** function is declared by **#pragma entry**.

(4) Initialization of Control Registers
The address of the variable vector table is written to INTB. FINTV, FPSW, BPC, BPSW, EXTB, and DPSW are also initialized as required. These registers can be initialized using the intrinsic functions of the compiler.
Note however that only PSW is not initialized because it holds the interrupt mask setting.

(5) Initializing the Trigonometric Function Unit
When the **-tfu** option is specified, also call the **__init_tfu()** intrinsic function to initialize the trigonometric func-tion unit.

(6) Initialization Processing of Sections

The initialization routine for RAM area sections (**_INITSCT**) is called. Uninitialized data sections are initialized to zero. For initialized data sections, the initial values of the ROM area are copied to the RAM area. **_INITSCT** is provided as a standard library.

The user needs to write the sections to be initialized to the tables for section initialization (**DTBL** and **BTBL**). The section address operator is used to set the start and end addresses of the sections used by the **_INITSCT** function.

Section names in the section initialization tables are declared, using **C$BSEC** for uninitialized data areas, and **C$DSEC** for initialized data areas.

A coding example is shown below.

Example:

```
#pragma section C C$DSEC     //Section name must be C$DSEC
extern const struct {
    void *rom_s;             //Start address member of the initialized data
                             //section in ROM
    void *rom_e;             //End address member of the initialized data
                             //section in ROM
    void *ram_s;             //Start address member of the initialized data
                             //section in RAM
} DTBL[] = {__sectop("D"), __secend("D"), __sectop("R")};

#pragma section C C$BSEC     //Section name must be C$BSEC
extern const struct {
    void *b_s;            //Start address member of the uninitialized data section
    void *b_e;              //End address member of the uninitialized data section
} BTBL[] = {__sectop("B"), __secend("B")};
```

(7) Initialization Processing of Libraries

The routine for performing necessary initialization processing (**_INITLIB**) is called when the C/C++ library functions are used.

In order to set only those values which are necessary for the functions that are actually to be used, please refer to the following guidelines.

- When an initial setting is required in the prepared low-level interface routines, the initial setting (**_INIT_LOWLEVEL**) in accordance with the specifications of the low-level interface routines is necessary.
- When using the **rand** function or **strtok** function, initial settings other than those for standard I/O (**_INIT_OTHERLIB**) are necessary.

An example of a program to perform initial library settings is shown below.

```
#include <stdio.h>
#include <stdlib.h>
#define IOSTREAM 3
const size_t _sbrk_size = 520;     // Specifies the minimum unit of the size to
                                   // define for the heap area (default: 1024)
extern char *_s1ptr;

#ifdef __cplusplus
extern "C" {
#endif
void _INITLIB (void)
{
    _INIT_LOWLEVEL();              // Set initial setting for low-level
                                   // interface routines
    _INIT_OTHERLIB();              // Set initial setting for rand function and
                                   // strtok function
}

void _INIT_LOWLEVEL (void)
{                                  // Set necessary initial setting for low-level
                                   // library
}

void _INIT_OTHERLIB(void)
{
    srand(1);                      // Set initial setting if using rand function
    _s1ptr=NULL;                   // Set initial setting if using strtok function
}
#ifdef __cplusplus
}
#endif
```

Notes 1. Specify the filename for the standard I/O file. This name is used in the low-level interface routine "**open**".

Notes 2. In the case of a console or other interactive device, a flag is set to prevent the use of buffering.

(8) Initialization of Global Class Objects
When developing a C++ program, the routine (**_CALL_INIT**) for calling the constructor of a class object that is declared as global is called. **_CALL_INIT** is provided as a standard library.

(9) Initialization of PSW for main Function Execution
The PSW register is initialized. The interrupt mask setting is canceled here.

(10) Changing of PM Bit in PSW
After a reset, operation is in privileged mode (PM bit in PSW is 0). To switch to user mode, intrinsic function **chg_pmusr** is executed.
When using the **chg_pmusr** function, some care should be taken. Refer to the description of **chg_pmusr** in 4.2.6 Intrinsic Functions.

(11) User Program Execution
The **main** function is executed.

(12) Global Class Object Postprocessing
When developing a C++ program, the routine (**_CALL_END**) for calling the destructor of a class object that is declared as global is called. **_CALL_END** is provided as a standard library.

8.3.3 Coding Example of Initial Setting Routine

A coding example of the **PowerOn_Reset_PC** function is shown here.
For the actual initial setting routine created in the integrated development environment, refer to 8.4 Coding Example.

```
#include     <machine.h>
#include     <_h_c_lib.h>
#include     "typedefine.h"
#include     "stacksct.h"

#ifdef __cplusplus
extern "C" {
#endif
void PowerOn_Reset_PC(void);
void main(void);
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus                     // Use SIM I/O
extern "C" {
#endif
extern void _INITLIB(void);
#ifdef __cplusplus
}
#endif

#define PSW_init  0x00010000
#define FPSW_DPSW_init 0x00000100

#pragma section ResetPRG
#pragma entry PowerOn_Reset_PC
void PowerOn_Reset_PC(void)
{
#if (__RX_ISA_VERSION__ >= 2) || defined(__RXV2)
    set_extb(__sectop("EXCEPTVECT"));
#endif
    set_intb(__sectop("C$VECT"));
#ifdef __FPU
    set_fpsw(FPSW_DPSW_init);
#ifdef __DPFPU
    __set_dpsw(FPSW_DPSW_init);
#endif
#endif
#if __TFU == 1
    __init_tfu();
#endif

    _INITSCT();
    _INITLIB();
    set_psw(PSW_init);
    main();
    brk();
}
```

8.3.4 Low-Level Interface Routines

Low-level interface routines are user-defined functions which are called from library functions in order to implement library functions in accordance with the specifications of the user system. They need to be created in any of the following cases.

(1) When using library functions related to standard I/O or memory management
(2) When a library function has to be a reentrant library

Table 8.2 lists the low-level interface routines used by C library functions.

Table 8.2 List of Low-Level Interface Routines

Name *1	Description
open	Opens file.
close	Closes file.
read	Reads from file.
write	Writes to file.
lseek	Sets the read/write position in a file.
sbrk	Allocates area in memory.
errno_addr *2	Acquires errno address.
wait_sem *2	Defines semaphore.
signal_sem *2	Releases semaphore.

Notes 1. The function names **open**, **close**, **read**, **write**, **lseek**, **sbrk**, **error_addr**, **wait_sem**, and **signal_sem** are reserved for low-level interface routines. They should not be used in user programs.

Notes 2. These routines are necessary when the reentrant library is used.

Initialization necessary for low-level interface routines must be performed on program startup. This initialization should be performed using the **_INIT_LOWLEVEL** function in library initial setting processing (**_INITLIB**).
Below, after explaining the basic approach to low-level I/O, the specifications for each interface routine are described.

(1) Approach to I/O
In the standard I/O library, files are managed by means of **FILE**-type data; but in low-level interface routines, positive integers are assigned in a one-to-one correspondence with actual files for management. These integers are called file numbers.
In the **open** routine, a file number is provided for a specified filename. The **open** routine must set the following information such that this number can be used for file input and output.

- The device type of the file (console, printer, disk file, etc.) (In the cases of special devices such as consoles or printers, special filenames must be set by the system and identified in the **open** routine.)
- When using file buffering, information such as the buffer position and size
- In the case of a disk file, the byte offset from the start of the file to the position for reading or writing

Based on the information set using the **open** routine, all subsequent I/O (**read** and **write** routines) and read/write positioning (**lseek** routine) is performed.
When output buffering is being used, the **close** routine should be executed to write the contents of the buffer to the actual file, so that the data area set by the **open** routine can be reused.

(2) Specifications of Low-Level Interface Routines
In this section, specifications for low-level interface routines are described. For each routine, the interface for calling the routine, its operation, and information for using the routine are described.
The interface for the routines is indicated using the following format. Low-level interface routines should always be given a prototype declaration. Add "**extern C**" to declare in the C++ program.

(Routine name)

[Description]

- (A summary of the routine operations is given)

[Return value]

Normal:	(The return value on normal termination is explained)
Error:	(The return value when an error occurs is given)

[Parameters]

(Name)	(Meaning)
(The name of the parameter appearing in the interface)	(The value passed as a parameter)

long open (const char *name, long mode, long flg)

[Description]

- Prepares for operations on the file corresponding to the filename of the first parameter. In the open routine, the file type (console, printer, disk file, etc.) must be determined in order to enable writing or reading at a later time. The file type must be referenced using the file number returned by the open routine each time reading or writing is to be performed.
- The second parameter mode specifies processing to be performed when the file is opened. The meanings of each of the bits of this parameter are as follows.

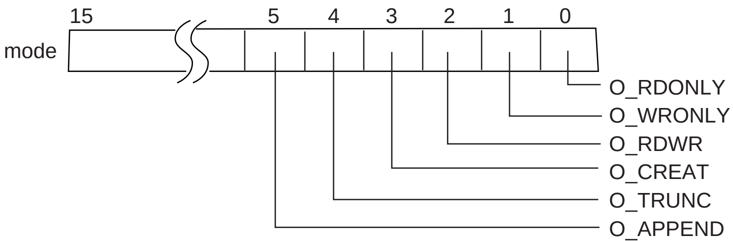


Table 8.3 Explanation of Bits in Parameter "mode" of open Routine

mode Bit	Description
O_RDONLY (bit 0)	When this bit is 1, the file is opened in read-only mode
O_WRONLY (bit 1)	When this bit is 1, the file is opened in write-only mode
O_RDWR (bit 2)	When this bit is 1, the file is opened for both reading and writing
O_CREAT (bit 3)	When this bit is 1, if a file with the filename given does not exist, it is created
O_TRUNC (bit 4)	When this bit is 1, if a file with the filename given exists, the file contents are deleted and the file size is set to 0
O_APPEND (bit 5)	Sets the position within the file for the next read/write operation When 0: Set to read/write from the beginning of file When 1: Set to read/write from file end

- When there is a contradiction between the file processing specified by mode and the properties of the actual file, error processing should be performed. When the file is opened normally, the file number (a positive integer) should be returned which should be used in subsequent read, write, lseek, and close routines. The correspondence between file numbers and the actual files must be managed by low-level interface routines. If the open operation fails, -1 should be returned.

[Return value]

Normal:	The file number for the successfully opened file
Error:	-1

[Parameters]

name	Name of the file
mode	Specifies the type of processing when the file is opened
flg	Specifies processing when the file is opened (always 0777)

long close (long fileno)

[Description]

- The file number obtained using the open routine is passed as a parameter.
- The file management information area set using the open routine should be released to enable reuse. Also, when output file buffering is performed in low-level interface routines, the buffer contents should be written to the actual file.
- When the file is closed successfully, 0 is returned; if the close operation fails, -1 is returned.

[Return value]

Normal:	0
Error:	-1

[Parameters]

fileno	Number of the file to be closed

long read (long fileno, unsigned char *buf, long count)

[Description]

- Data is read from the file specified by the first parameter (fileno) to the area in memory specified by the second parameter (buf). The number of bytes of data to be read is specified by the third parameter (count).
- When the end of the file is reached, only a number of bytes fewer than or equal to count bytes can be read.
- The position for file reading/writing advances by the number of bytes read.
- When reading is performed successfully, the actual number of bytes read is returned; if the read operation fails, -1 is returned.

[Return value]

Normal:	Actual number of bytes read
Error:	-1

[Parameters]

fileno	Number of the file to be read
buf	Memory area to store read data
count	Number of bytes to read

long write (long fileno, const unsigned char *buf, long count)

[Description]

- Writes data to the file indicated by the first parameter (fileno) from the memory area indicated by the second parameter (buf). The number of bytes to be written is indicated by the third parameter (count).
- If the device (disk, etc.) of the file to be written is full, only a number of bytes fewer than or equal to count bytes can be written. It is recommended that, if the number of bytes actually written is zero a certain number of times in succession, the disk should be judged to be full and an error (-1) should be returned.
- The position for file reading/writing advances by the number of bytes written. If writing is successful, the actual number of bytes written should be returned; if the write operation fails, -1 should be returned.

[Return value]

Normal:	Actual number of bytes written
Error:	-1

[Parameters]

fileno	Number of the file to which data is to be written
buf	Memory area containing data for writing
count	Number of bytes to write

long lseek (long fileno, long offset, long base)

[Description]

- Sets the position within the file, in byte units, for reading from and writing to the file.
- The position within a new file should be calculated and set using the following methods, depending on the third parameter (base).
  (1) When base is 0: Set the position at offset bytes from the file beginning
  (2) When base is 1: Set the position at the current position plus offset bytes
  (3) When base is 2: Set the position at the file size plus offset bytes
- When the file is a console, printer, or other interactive device, when the new offset is negative, or when in cases (1) and (2) the file size is exceeded, an error occurs.
- When the file position is set correctly, the new position for reading/writing should be returned as an offset from the file beginning; when the operation is not successful, -1 should be returned.

[Return value]

Normal:	The new position for file reading/writing, as an offset in bytes from the file beginning
Error:	-1

[Parameters]

fileno	File number
offset	Position for reading/writing, as an offset (in bytes)
base	Starting-point of the offset

char *sbrk (size_t size)

[Description]

- The size of the memory area to be allocated is passed as a parameter.
- When calling the sbrk routine several times, memory areas should be allocated in succession starting from lower addresses. If the memory area for allocation is insufficient, an error should occur. When allocation is successful, the address of the beginning of the allocated memory area should be returned; if unsuccessful, "(char *) -1" should be returned.
- If you wish to use the standard library function malloc, calloc, or realloc, or the C++ function new, allocate at least 16 bytes of memory.

[Return value]

Normal:	Start address of allocated memory
Error:	(char *) -1

[Parameters]

size	Size of data to be allocated

long *errno_addr (void)

[Description]

- Returns the address of the error number of the current task.
- This routine is necessary when using a standard library, which was created by the standard library generator with the reent option specified.

[Return value]

Address of the error number of the current task

long wait_sem (long semnum)

[Description]

- Defines the semaphore specified by semnum.
- When the semaphore has been defined normally, 1 must be returned. Otherwise, 0 must be returned.
- This routine is necessary when using a standard library, which was created by the standard library generator with the reent option specified.

[Return value]

Normal:	1
Error:	0

[Parameters]

semnum	Semaphore ID

long signal_sem (long semnum)

[Description]

- Releases the semaphore specified by semnum.
- When the semaphore has been released normally, 1 must be returned. Otherwise, 0 must be returned.
- This routine is necessary when using a standard library, which was created by the standard library generator with the reent option specified.

[Return value]

Normal:	1
Error:	0

[Parameters]

semnum	Semaphore ID

(3) Example of Coding Low-Level Interface Routines

```
/***********************************************************************/
/*                              lowsrc.c:                              */
/*---------------------------------------------------------------------*/
/*              RX Family Simulator Debugger Interface Routine         */
/*       - Supports only the standard input/output(stdin,stdout,stderr) -    */
/***********************************************************************/
#include <string.h>

/* File Numbers */
#define STDIN  0                    /* Standard Input (Console) */
#define STDOUT 1                    /* Standard Output (Console) */
#define STDERR 2                    /* Standard Error Output (Console) */

#define FLMIN 0                     /* Minimum value of the File Number */
#define FLMAX 3                     /* Maximum value of the Number of Files */

/* File Flags */
#define O_RDONLY 0x0001             /* Read Only */
#define O_WRONLY 0x0002             /* Write Only */
#define O_RDWR 0x0004               /* Read and Write */

/* Special Character Codes */
#define CR 0x0d                     /* Carriage Return */
#define LF 0x0a                     /* Line Feed */

/* Heap Size of the sbrk */
#define HEAPSIZE 1024

/***********************************************************************/
/*                    Declaration of Using Functions                   */
/*        - Outputs and Inputs to a Console on a Simulator Debugger -  */
/***********************************************************************/
extern void charput(char);               /* Inputs a Byte */
extern char charget(void);               /* Outputs a Byte */

/***********************************************************************/
/*                    Definition of Static Variables                   */
/*                - Used by the Low-Level Interface Routine -          */
/***********************************************************************/
char flmod[FLMAX];                       /* Open File Modes */

union HEAP_TYPE{
    long dummy;                          /* (Dummy: for 4-bytes alignment) */
    char heap[HEAPSIZE];                 /* Heap Area of the sbrk */
};

static union HEAP_TYPE heap_area;

static char *brk=(char*)&heap_area;      /* Latest Address of sbrk Assigned */
```

```
/******************************************************************************/
/*                      open --- Open A File                                  */
/*                 Return Value: File Number (Success)                        */
/*                                    -1      (Fail)                          */
/******************************************************************************/
long open(const char *name,                       /* File Name */
          long mode,                              /* File Open Mode */
          long flg)                               /* Open Flag (Not Used) */
{
    /* Checks mode of the file, and Returns file number */

    if (strcmp(name,"stdin")==0) {            /* Standard Input File */
        if ((mode&O_RDONLY)==0) {
            return (-1);
        }
        flmod[STDIN]=mode;
        return (STDIN);
    }

    else if (strcmp(name,"stdout")==0) {      /* Standard Output File */
        if ((mode&O_WRONLY)==0) {
            return (-1);
        }
        flmod[STDOUT]=mode;
        return (STDOUT);
    }

    else if (strcmp(name,"stderr")==0){       /* Standard Error Output File */
        if ((mode&O_WRONLY)==0) {
            return (-1);
        }
        flmod[STDERR]=mode;
        return (STDERR);
    }

    else {
        return (-1);                          /* Error */
    }
}

/******************************************************************************/
/*                      close --- Close A File                                */
/*                     Return Value: 0    (Success)                           */
/*                                  -1    (Fail)                              */
/******************************************************************************/
long close(long fileno)                       /* File Number */
{
    if (fileno<FLMIN || FLMAX<fileno) {       /* Checks the File Number */
        return -1;
    }

    flmod[fileno]=0;                          /* Resets the File Mode */

    return 0;
}
```

```
/*****************************************************************************/
/*                          read --- Input Data                              */
/*                Return Value: Bytes Number of Read  (Success)              */
/*                              -1                    (Fail)                 */
/*****************************************************************************/
long read(long fileno,                        /* File Number */
         unsigned char *buf,                  /* Write Buffer Address */
         long  count)                         /* Bytes Number of Read */
{
    unsigned long i;

    /*  Checks mode of the file, and Sets the Write Buffer each bytes */

    if (flmod[fileno]&O_RDONLY || flmod[fileno]&O_RDWR) {
        for (i=count;i>0;i--) {
            *buf=charget();
            if (*buf==CR) {                   /* Replaces CR into LF */
                *buf=LF;
            }
        buf++;
        }
        return count;
    }

    else {
        return -1;
    }
}

/*****************************************************************************/
/*                         write --- Output Data                             */
/*                Return Value: Bytes Number of Write (Success)              */
/*                              -1                    (Fail)                 */
/*****************************************************************************/
long write(long fileno,                       /* File Number */
          const unsigned char *buf,           /* Read Buffer Address */
          long count)                         /* Bytes Number of Write */
{
    unsigned long i;
    unsigned char c;

    /*  Checks mode of the file, and Output from the Rrite Buffer each bytes */

    if (flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR) {
        for (i=count; i>0; i--) {
            c=*buf++;
            charput(c);
        }
        return count;
    }

    else {
        return -1;
    }
}
```

```
/*****************************************************************************/
/*                lseek --- Sets Position of Reading and Writing            */
/*              Return Value: Offset of the File Position (Success)         */
/*                             -1                            (Fail)         */
/*              (lseek doesn't support Console Input/Output)                */
/*****************************************************************************/
long lseek(long fileno,                    /* File Number */
           long offset,                    /* Position of Reading and Writing */
           long  base)                     /* Start of Offset */
{
    return -1;
}

/*****************************************************************************/
/*                    sbrk --- Allocate Heap Memory                         */
/*              Return Value: Top address of Allocated Area (Success)       */
/*                             -1                              (Fail)       */
/*****************************************************************************/
char *sbrk(size_t size)                    /* Allcation Memory Size */
{
    char *p;

    /* Checks Free Area */

    if (brk+size>heap_area.heap+HEAPSIZE) {
        return (char *)-1;
    }

    p=brk;                                 /* Allocate an Area */
    brk+=size;                             /* Updates the Latest Address */
    return p;
}
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          lowlvl.src                                  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  RX Family Simulator/Debugger Interface Routine                      ;
;             - Inputs and outputs one character -                     ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
         .GLB    _charput
         .GLB    _charget

SIM_IO   .EQU 0h

         .SECTION   P,CODE
;-----------------------------------------------------------------------
;  _charput:
;-----------------------------------------------------------------------
_charput:
         MOV.L      #IO_BUF,R2
         MOV.B      R1,[R2]
         MOV.L      #1220000h,R1
         MOV.L      #PARM,R3
         MOV.L      R2,[R3]
         MOV.L      R3,R2
         MOV.L      #SIM_IO,R3
         JSR        R3
         RTS

;-----------------------------------------------------------------------
;  _charget:
;-----------------------------------------------------------------------
_charget:
        MOV.L       #1210000h,R1
        MOV.L       #IO_BUF,R2
        MOV.L       #PARM,R3
        MOV.L       R2,[R3]
        MOV.L       R3,R2
        MOV.L       #SIM_IO,R3
        JSR         R3
        MOV.L       #IO_BUF,R2
        MOVU.B      [R2],R1
        RTS

;-----------------------------------------------------------------------
;  I/O Buffer
;-----------------------------------------------------------------------
         .SECTION  B,DATA,ALIGN=4
PARM:    .BLKL     1
         .SECTION  B_1,DATA
IO_BUF:  .BLKB     1
         .END
```

(4) Example of Low-Level Interface Routine for Reentrant Library

The following shows an example of low-level interface routines for a reentrant library. These routines are necessary when using a library, which was created by the library generator with the **reent** option specified.

When failing to allocate semaphores with the **wait_sem** function or **signal_sem** function, set **errno** as follows to return from the library function.

Table 8.4 Error number list of the reentrant library sets to errno variable

Function Name	errno	Description
wait_sem	EMALRESM	Failed to allocate semaphore resources for malloc.
	ETOKRESM	Failed to allocate semaphore resources for strtok.
	EFLSRESM	Failed to allocate semaphore resources for _Files.
	EMBLNRESM	Failed to allocate semaphore resources for mbrlen.
signal_sem	EMALFRSM	Failed to release semaphore resources for malloc.
	ETOKFRSM	Failed to release semaphore resources for strtok.
	EFLSFRSM	Failed to release semaphore resources for _Files.
	EMBLNFRSM	Failed to release semaphore resources for mbrlen.

When an interrupt with a priority level higher than the current level is generated after semaphores have been defined, dead locks will occur if semaphores are defined again. Therefore, be careful for processes that share resources because they might be nested by interrupts.

```
#define MALLOC_SEM   1  /* Semaphore No. for malloc */
#define STRTOK_SEM   2  /* Semaphore No. for strtok */
#define FILE_TBL_SEM 3  /* Semaphore No. for fopen  */
#define MBRLEN_SEM   4  /* Semaphore No. for mbrlen */
#define FPSWREG_SEM  5  /* Semaphore No. for FPSW register */
#define FILES_SEM    6  /* Semaphore No. for _Files */
#define SEMSIZE     26  /* FILES_SEM + _nfiles (assumed _nfiles=20) */

#define TRUE 1
#define FALSE 0
#define OK 1
#define NG 0
extern long *errno_addr(void);
extern long wait_sem(long);
extern long signal_sem(long);
static long sem_errno;
static int force_fail_signal_sem = FALSE;
static int semaphore[SEMSIZE];

/******************************************************************************/
/*                  errno_addr: Acquisition of errno address                  */
/*                        Return value: errno address                         */
/******************************************************************************/
long *errno_addr(void)
{
    /* Return the errno address of the current task */
    return (&sem_errno);
}

/******************************************************************************/
/*           wait_sem: Defines the specified numbers of semaphores            */
/*                       Return value: OK(=1) (Normal)                        */
/*                                     NG(=0) (Error)                         */
/******************************************************************************/
long wait_sem(long semnum) /* Semaphore ID */
{
    if((0 < semnum) && (semnum < SEMSIZE)) {
        if(semaphore[semnum] == FALSE) {
            semaphore[semnum] = TRUE;
            return(OK);
        }
    }
    return(NG);
}

/******************************************************************************/
/*          signal_sem: Releases the specified numbers of semaphores          */
/*                       Return value: OK(=1) (Normal)                        */
/*                                     NG(=0) (Error)                         */
/******************************************************************************/
long signal_sem(long semnum) /* Semaphore ID */
{
    if(!force_fail_signal_sem) {
        if((0 <= semnum) && (semnum < SEMSIZE)) {
            if( semaphore[semnum] == TRUE ) {
                semaphore[semnum] = FALSE;
                return(OK);
            }
        }
    }
    return(NG);
}
```

8.3.5 Termination Processing Routine

(1) Example of Preparation of a Routine for Termination Processing Registration and Execution (atexit)
The method for preparation of the library function **atexit** to register termination processing is described.
The **atexit** function registers, in a table for termination processing, a function address passed as a parameter. If the number of functions registered exceeds the limit (in this case, the number that can be registered is assumed to be 32), or if an attempt is made to register the same function twice, **NULL** is returned. Otherwise, a value other than **NULL** (in this case, the address of the registered function) is returned.
A program example is shown below.
Example:

```
#include <stdlib.h>


long _atexit_count=0 ;

void (*_atexit_buf[32])(void) ;

#ifdef __cplusplus
extern "C"
#endif
long atexit(void (*f)(void))
{
    int i;

    for(i=0; i<_atexit_count ; i++)    // Check whether it is already registered
        if(_atexit_buf[i]==f)
             return 1;
    if(_atexit_count==32)      // Check the limit value of number of registration
        return 1;
    else {
        atexit_buf[_atexit_count++]=f; // Register the function address
        return 0;
    }
}
```

(2) Example of Preparation of a Routine for Program Termination (exit)
The method for preparation of an **exit** library function for program termination is described. Program termination processing will differ among user systems; refer to the program example below when preparing a termination procedure according to the specifications of the user system.
The **exit** function performs termination processing for a program according to the termination code for the program passed as a parameter, and returns to the environment in which the program was started. Here, the termination code is set to an external variable, and execution returned to the environment saved by the **setjmp** function immediately before the **main** function was called. In order to return to the environment prior to program execution, the following **callmain** function should be created, and instead of calling the function **main** from the **PowerOn_Reset_PC** initial setting function, the **callmain** function should be called.
A program example is shown below.

```
#include <setjmp.h>
#include <stddef.h>


extern long _atexit_count ;
extern void_t (*_atexit_buf[32])(void) ;
#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
int main(void);
extern jmp_buf _init_env ;
int _exit_code ;

#ifdef __cplusplus
extern "C"
#endif
void exit(int code)
{
  int i;
  _exit_code=code ;                  // Set the return code in _exit_code
  for(i=_atexit_count-1; i>=0; i--)// Execute in sequence the functions
     (*_atexit_buf[i])();            // registered by the atexit function
  _CLOSEALL();                       // Close all open functions
  longjmp(_init_env, 1) ;            // Return to the environment saved by
                                     // setjmp
}
#ifdef __cplusplus
extern "C"
#endif
void callmain(void)
{
     // Save the current environment using setjmp and call the main function
     if(!setjmp(_init_env))
         _exit_code=main();          // On  returning from the exit function,
                                     // terminate processing
}
```

(3) Example of Creation of an Abnormal Termination (abort) Routine

On abnormal termination, processing for abnormal termination must be executed in accordance with the specifications of the user system.
In a C++ program, the **abort** function will also be called in the following cases:

- When exception processing was unable to operate correctly.
- When a pure virtual function is called.
- When **dynamic_cast** has failed.
- When **typeid** has failed.
- When information could not be acquired when a class array was deleted.
- When the definition of the destructor call for objects of a given class causes a contradiction.

Below is shown an example of a program which outputs a message to the standard output device, then closes all files and begins an infinite loop to wait for reset.

```
#include <stdio.h>
#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
#ifdef __cplusplus
extern "C"
#endif
void abort(void)
{
        printf("program is abort !!\n"); //Output message
        _CLOSEALL();                      //Close all files
        while(1)    ;                     //Begin infinite loop
}
```

8.4 Coding Example

This section shows an example of an actual startup program created for the simulator in the integrated development environment when the RX610 is selected as the CPU type.

(1) Source Files
The startup program consists of the files shown in Table 8.5.

Table 8.5 List of Programs Created in Integrated Development Environment

	File Name	Description
(a)	resetprg.c	Initial setting routine (reset vector function)
(b)	intprg.c	Vector function definitions
(c)	vecttbl.c	Fixed vector table *1
(d)	dbsct.c	Section initialization processing (table)
(e)	lowsrc.c	Low-level interface routine (C language part)
(f)	lowlvl.src	Low-level interface routine (assembly language part)
(g)	sbrk.c	Low-level interface routine (**sbrk** function)
(h)	typedefine.h	Type definition header
(i)	vect.h	Vector function header
(j)	stacksct.h	Stack size settings
(k)	lowsrc.h	Low-level interface routine (C language header)
(l)	sbrk.h	Low-level interface routine (**sbrk** function header)

Notes 1. This is for the RXv1 instruction-set architecture.
For an RX instruction-set architecture other than the RXv1 instruction-set architecture, this becomes the "exception vector table".

The following shows the contents of files (a) to (l).

(a) resetprg.c: Initial Setting Routine (Reset Vector Function)

```
#include machine.h>
#include <_h_c_lib.h>
//#include <stddef.h>     // Remove the comment when you use errno
//#include <stdlib.h>     // Remove the comment when you use rand()
#include "typedefine.h"   // Define Types
#include "stacksct.h"     // Stack Sizes (Interrupt and User)

#ifdef __cplusplus        // For Use Reset vector
extern "C" {
#endif
void PowerOn_Reset_PC(void);
void main(void);
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus        // For Use SIM I/O
extern "C" {
#endif
extern void _INIT_IOLIB(void);
extern void _CLOSEALL(void);
#ifdef __cplusplus
}
#endif

#define PSW_init       0x00010000  // PSW bit pattern
#define FPSW_DPSW_init 0x00000000  // FPSW/DPSW bit base pattern

//extern void srand(_UINT);   // Remove the comment when you use rand()
//extern _SBYTE *_s1ptr;      // Remove the comment when you use strtok()

//#ifdef __cplusplus          // Use Hardware Setup
//extern "C" {
//#endif
//extern void HardwareSetup(void);
//#ifdef __cplusplus
//}
//#endif

//#ifdef __cplusplus         // Remove the comment when you use global class
object
//extern "C" {               // Sections C$INIT and C$END will be generated
//#endif
//extern void _CALL_INIT(void);
//extern void _CALL_END(void);
//#ifdef __cplusplus
//}
//#endif

#pragma section ResetPRG     // output PowerOn_Reset_PC to PResetPRG section

#pragma entry PowerOn_Reset_PC

void PowerOn_Reset_PC(void)
{
#if (__RX_ISA_VERSION__ >= 2) || defined(__RXV2)
   set_extb(__sectop(”EXCEPTVECT”)); // Remove the comment when you want to set
an address
                                    // into the Exception Vector Table Register
of RXv2 or later
```

```
#endif
    set_intb(__sectop("C$VECT"));
#ifdef __FPU
#ifdef __ROZ                 // Initialize FPSW/DPSW
#define _ROUND 0x00000001    // Let FPSW/DPSW RM/DRM bits=01 (round to zero)
#else
#define _ROUND 0x00000000    // Let FPSW/DPSW RM/DRM bits=00 (round to nearest)
#endif
#ifdef __DOFF
#define _DENOM 0x00000100    // Let FPSW/DPSW DN/DDN bit=1 (denormal as zero)
#else
#define _DENOM 0x00000000    // Let FPSW/DPSW DN/DDN bit=0 (denormal as is)
#endif
    set_fpsw(FPSW_DPSW_init | _ROUND | _DENOM);
#ifdef __DPFPU
     __set_dpsw(FPSW_DPSW_init | _ROUND | _DENOM);
#endif
#endif

_INITSCT();                 // Initialize Sections

_INIT_IOLIB();              // Use SIM I/O

//errno=0;                  // Remove the comment when you use errno
//srand((_UINT)1);          // Remove the comment when you use rand()
//_s1ptr=NULL;              // Remove the comment when you use strtok()

//HardwareSetup();          // Use Hardware Setup

//_CALL_INIT();            // Remove the comment when you use global class object

set_psw(PSW_init);          // Set Ubit & Ibit for PSW
//chg_pmusr();              // Remove the comment when you need to change PSW
// PMbit (SuperVisor->User)

main();

_CLOSEALL();                // Use SIM I/O

//_CALL_END();              // Remove the comment when you use global class
// object

brk();
}
```

(b) intprg.c: Vector Function Definitions

```
#include <machine.h>
#include "vect.h"
#pragma section IntPRG

// Exception (Supervisor Instruction)
void Excep_SuperVisorInst(void){/* brk(); */}

// Exception (Undefined Instruction)
void Excep_UndefinedInst(void){/* brk(); */}

// Exception (Floating Point)
void Excep_FloatingPoint(void){/* brk(); */}

// NMI
void NonMaskableInterrupt(void){/* brk(); */}

// Dummy
void Dummy(void){/* brk(); */}

// BRK
void Excep_BRK(void){ wait(); }
```

(c) vecttbl.c: Fixed Vector Table

```
#include "vect.h"

#pragma section C FIXEDVECT

void (*const Fixed_Vectors[])(void) = {
//;0xffffffd0  Exception (Supervisor Instruction)
    Excep_SuperVisorInst,
//;0xffffffd4  Reserved
    Dummy,
//;0xffffffd8  Reserved
    Dummy,
//;0xffffffdc  Exception (Undefined Instruction)
    Excep_UndefinedInst,
//;0xffffffe0  Reserved
    Dummy,
//;0xffffffe4  Exception (Floating Point)
    Excep_FloatingPoint,
//;0xffffffe8  Reserved
    Dummy,
//;0xffffffec  Reserved
    Dummy,
//;0xfffffff0  Reserved
    Dummy,
//;0xfffffff4  Reserved
    Dummy,
//;0xfffffff8  NMI
    NonMaskableInterrupt,
//;0xfffffffc  RESET
//;<<VECTOR DATA START (POWER ON RESET)>>
//;Power On Reset PC
PowerOn_Reset_PC
//;<<VECTOR DATA END (POWER ON RESET)>>
};
```

[Reference]
Case when an RX instruction-set architecture other than the RXv1 instruction-set architecture is selected (exception vector table)

```
#include "vect.h"

#pragma section C EXCEPTVECT

void (*const Excep_Vectors[])(void) = {
//;0xffffff80  Reserved
    Dummy,
//;0xffffff84  Reserved
    Dummy,
//;0xffffff88  Reserved
    Dummy,
//;0xffffff8c  Reserved
    Dummy,
//;0xffffff90  Reserved
    Dummy,
//;0xffffff94  Reserved
    Dummy,
//;0xffffff98  Reserved
    Dummy,
//;0xffffff9c  Reserved
    Dummy,
//;0xffffffa0  Reserved
    Dummy,
//;0xffffffa4  Reserved
    Dummy,
//;0xffffffa8  Reserved
    Dummy,
//;0xffffffac  Reserved
    Dummy,
//;0xffffffb0  Reserved
    Dummy,
//;0xffffffb4  Reserved
    Dummy,
//;0xffffffb8  Reserved
    Dummy,
//;0xffffffbc  Reserved
    Dummy,
//;0xffffffc0  Reserved
    Dummy,
//;0xffffffc4  Reserved
    Dummy,
//;0xffffffc8  Reserved
    Dummy,
//;0xffffffcc  Reserved
    Dummy,
//;0xffffffd0  Exception(Supervisor Instruction)
    Excep_SuperVisorInst,
//;0xffffffd4  Exception(Access Instruction)
    Excep_AccessInst,
//;0xffffffd8  Reserved
    Dummy,
//;0xffffffdc  Exception(Undefined Instruction)
    Excep_UndefinedInst,
//;0xffffffe0  Reserved
    Dummy,
//;0xffffffe4  Exception(Floating Point)
    Excep_FloatingPoint,
```

```
//;0xffffffe8  Reserved
    Dummy,
//;0xffffffec  Reserved
    Dummy,
//;0xfffffff0  Reserved
    Dummy,
//;0xfffffff4  Reserved
    Dummy,
//;0xfffffff8  NMI
    NonMaskableInterrupt,
};

#pragma section C RESETVECT

void (*const Reset_Vectors[])(void) = {
//;0xfffffffc  RESET
    PowerON_Reset_PC
};
```

(d) dbsct.c: Section Initialization Processing (table)

```
#include "typedefine.h"

#pragma unpack

#pragma section C C$DSEC
extern const struct {
    _UBYTE *rom_s;    /* Start address of the initialized data section in ROM */
    _UBYTE *rom_e;    /* End address of the initialized data section in ROM   */
    _UBYTE *ram_s;    /* Start address of the initialized data section in RAM */
}   _DTBL[] = {
    { __sectop("D"), __secend("D"), __sectop("R") },
    { __sectop("D_2"), __secend("D_2"), __sectop("R_2") },
    { __sectop("D_1"), __secend("D_1"), __sectop("R_1") }
};
#pragma section C C$BSEC
extern const struct {
    _UBYTE *b_s;         /* Start address of non-initialized data section */
    _UBYTE *b_e;         /* End address of non-initialized data section */
}   _BTBL[] = {
    { __sectop("B"), __secend("B") },
    { __sectop("B_2"), __secend("B_2") },
    { __sectop("B_1"), __secend("B_1") }
};

#pragma section

/*
** CTBL prevents excessive output of W0561100 messages when linking.
** Even if CTBL is deleted, the operation of the program does not change.
*/
_UBYTE * const _CTBL[] = {
    __sectop("C_1"), __sectop("C_2"), __sectop("C"),
    __sectop("W_1"), __sectop("W_2"), __sectop("W")
};

#pragma packoption
```

(e) lowsrc.c : Low-Level Interface Routine (C Language Part)

```
#include <string.h>
#include <stdio.h>
#include <stddef.h>
#include "lowsrc.h"

#define STDIN  0
#define STDOUT 1
#define STDERR 2

#define FLMIN  0
#define _MOPENR0x1
#define _MOPENW0x2
#define _MOPENA0x4
#define _MTRUNC0x8
#define _MCREAT0x10
#define _MBIN0x20
#define _MEXCL0x40
#define _MALBUF0x40
#define _MALFIL0x80
#define _MEOF0x100
#define _MERR0x200
#define _MLBF0x400
#define _MNBF0x800
#define _MREAD0x1000
#define _MWRITE0x2000
#define _MBYTE0x4000
#define _MWIDE0x8000

#define O_RDONLY0x0001
#define O_WRONLY0x0002
#define O_RDWR0x0004
#define O_CREAT0x0008
#define O_TRUNC0x0010
#define O_APPEND0x0020

#define CR 0x0d
#define LF 0x0a

extern const long _nfiles;
char flmod[IOSTREAM];

unsigned char sml_buf[IOSTREAM];

#define FPATH_STDIN     "C:\\stdin"
#define FPATH_STDOUT    "C:\\stdout"
#define FPATH_STDERR    "C:\\stderr"

extern void charput(unsigned char);
extern unsigned char charget(void);

#include <stdio.h>
FILE *_Files[IOSTREAM];
char *env_list[] = {
    "ENV1=temp01",
    "ENV2=temp02",
    "ENV9=end",
    '\0'
};
```

```
char **environ = env_list;

void _INIT_IOLIB( void )
{
    _Files[0] = stdin;
    _Files[1] = stdout;
    _Files[2] = stderr;

    if( freopen( FPATH_STDIN, "r", stdin ) == NULL )
        stdin->_Mode = 0xffff;
    stdin->_Mode  = _MOPENR;
    stdin->_Mode |= _MNBF;
    stdin->_Bend = stdin->_Buf + 1;

    if( freopen( FPATH_STDOUT, "w", stdout ) == NULL )
        stdout->_Mode = 0xffff;
    stdout->_Mode |= _MNBF;
    stdout->_Bend = stdout->_Buf + 1;

    if( freopen( FPATH_STDERR, "w", stderr ) == NULL )
        stderr->_Mode = 0xffff;
    stderr->_Mode |= _MNBF;
    stderr->_Bend = stderr->_Buf + 1;
}

void _CLOSEALL( void )
{
    long i;
    for( i=0; i < _nfiles; i++ )
    {
        if( _Files[i]->_Mode & (_MOPENR | _MOPENW | _MOPENA ) )
        fclose( _Files[i] );
    }
}

long open(const char *name,
     long  mode,
     long  flg)
{

    if( strcmp( name, FPATH_STDIN ) == 0 )
    {
        if( ( mode & O_RDONLY ) == 0 ) return -1;
        flmod[STDIN] = mode;
        return STDIN;
    }
    else if( strcmp( name, FPATH_STDOUT ) == 0 )
    {
        if( ( mode & O_WRONLY ) == 0 ) return -1;
        flmod[STDOUT] = mode;
        return STDOUT;
    }
    else if(strcmp(name, FPATH_STDERR ) == 0 )
    {
        if( ( mode & O_WRONLY ) == 0 ) return -1;
        flmod[STDERR] = mode;
        return STDERR;
    }
    else return -1;
}
```

```
long close( long fileno )
{
    return 1;
}

long write(long  fileno,
      const unsigned char *buf,
      long  count)
{
    long    i;
    unsigned char    c;

    if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR)
    {
        if( fileno == STDIN ) return -1;
        else if( (fileno == STDOUT) || (fileno == STDERR) )
        {
            for( i = count; i > 0; --i )
            {
                c = *buf++;
                charput(c);
            }
            return count;
        }
        else return -1;
    }
    else return -1;
}

long read( long fileno, unsigned char *buf, long count )
{
   long i;
       if((flmod[fileno]&_MOPENR) || (flmod[fileno]&O_RDWR)){
             for(i = count; i > 0; i--){
                   *buf = charget();
                   if(*buf==CR){
                         *buf = LF;
                   }
                   buf++;
             }
             return count;
       }
       else {
             return -1;
       }
}

long lseek( long fileno, long offset, long base )
{
    return -1L;
}
```

(f) lowlvl.src: Low-Level Interface Routine (Assembly Language Part)

```
          .GLB     _charput
          .GLB     _charget

SIM_IO    .EQU 0h

          .SECTION    P,CODE
;-----------------------------------------------------------------------
;  _charput:
;-----------------------------------------------------------------------
_charput:
          MOV.L       #IO_BUF,R2
          MOV.B       R1,[R2]
          MOV.L       #1220000h,R1
          MOV.L       #PARM,R3
          MOV.L       R2,[R3]
          MOV.L       R3,R2
          MOV.L       #SIM_IO,R3
          JSR         R3
          RTS

;-----------------------------------------------------------------------
;  _charget:
;-----------------------------------------------------------------------
_charget:
         MOV.L        #1210000h,R1
         MOV.L        #IO_BUF,R2
         MOV.L        #PARM,R3
         MOV.L        R2,[R3]
         MOV.L        R3,R2
         MOV.L        #SIM_IO,R3
         JSR          R3
         MOV.L        #IO_BUF,R2
         MOVU.B       [R2],R1
         RTS

;-----------------------------------------------------------------------
;  I/O Buffer
;-----------------------------------------------------------------------
          .SECTION  B,DATA,ALIGN=4
PARM:     .BLKL     1
          .SECTION  B_1,DATA
IO_BUF:   .BLKB     1
          .END
```

(g) sbrk.c: Low-Level Interface Routine (sbrk Function)

```
#include <stddef.h>
#include <stdio.h>
#include "typedefine.h"
#include "sbrk.h"

_SBYTE  *sbrk(size_t size);

//const size_t _sbrk_size=    /* Specifies the minimum unit of */
                              /* the defined heap area*/

extern _SBYTE *_s1ptr;

union HEAP_TYPE {
    _SDWORD  dummy ;          /* Dummy for 4-byte boundary */
    _SBYTE heap[HEAPSIZE];    /* Declaration of the area managed by sbrk */
};

static union HEAP_TYPE heap_area ;

/* End address allocated by sbrk */
static _SBYTE *brk=(_SBYTE *)&heap_area;

/**********************************************************************/
/*      sbrk:Memory area allocation                                   */
/*           Return value:Start address of allocated area (Pass)      */
/*                        -1                             (Failure)    */
/**********************************************************************/
_SBYTE  *sbrk(size_t size)                      /* Assigned area size   */
{
    _SBYTE  *p;

    if(brk+size > heap_area.heap+HEAPSIZE){     /* Empty area size      */
        p = (_SBYTE *)-1;
    }
    else {
        p = brk;                                /* Area assignment      */
        brk += size;                            /* End address update   */
    }
    return p;
}
```

(h) typedefine.h: Type Definition Header

```
typedef signed char _SBYTE;
typedef unsigned char _UBYTE;
typedef signed short _SWORD;
typedef unsigned short _UWORD;
typedef signed int _SINT;
typedef unsigned int _UINT;
typedef signed long _SDWORD;
typedef unsigned long _UDWORD;
typedef signed long long _SQWORD;
typedef unsigned long long _UQWORD;
```

(i) vect.h: Vector Function Header

```
// Exception (Supervisor Instruction)
#pragma interrupt (Excep_SuperVisorInst)
void Excep_SuperVisorInst(void);

// Exception (Undefined Instruction)
#pragma interrupt (Excep_UndefinedInst)
void Excep_UndefinedInst(void);

// Exception (Floating Point)
#pragma interrupt (Excep_FloatingPoint)
void Excep_FloatingPoint(void);

// NMI
#pragma interrupt (NonMaskableInterrupt)
void NonMaskableInterrupt(void);

// Dummy
#pragma interrupt (Dummy)
void Dummy(void);

// BRK
#pragma interrupt (Excep_BRK(vect=0))
void Excep_BRK(void);

//;<<VECTOR DATA START (POWER ON RESET)>>
//;Power On Reset PC
extern void PowerOn_Reset_PC(void);
//;<<VECTOR DATA END (POWER ON RESET)>>
```

(j) stacksct.h: Stack Size Settings

```
// #pragma stacksize su=0x100 // Remove the comment when you use user stack
#pragma stacksize si=0x300
```

(k) lowsrc.h: Low-Level Interface Routine (C Language Header)

```
/*Number of I/O Streams*/
#define IOSTREAM 20
```

(l) sbrk.h: Low-Level Interface Routine (sbrk Function Header)

```
/* Size of area managed by sbrk */
#define HEAPSIZE 0x400
```

(2) Execution Commands
The following shows an example of commands for building these files.
In this example, the name of the user program file (containing the **main** function) is UserProgram.c, and the body of the file names (names excluding extensions) for the load module or library to be created is LoadModule.

```
lbgrx -isa=rxv1 -output=LoadModule.lib
ccrx -isa=rxv1 -output=obj UserProgram.c
ccrx -isa=rxv1 -output=obj resetprg.c
ccrx -isa=rxv1 -output=obj intprg.c
ccrx -isa=rxv1 -output=obj vecttbl.c
ccrx -isa=rxv1 -output=obj dbsct.c
ccrx -isa=rxv1 -output=obj lowsrc.c
asrx -isa=rxv1 lowlvl.src
ccrx -isa=rxv1 -output=obj sbrk.c
rlink -rom=D=R,D_1=R_1,D_2=R_2 -list=LoadModule.map
-start=B_1,R_1,B_2,R_2,B,R,SI/01000,PResetPRG/
0FFFF8000,C_1,C_2,C,C$*,D_1,D_2,D,P,PIntPRG,W*,L/0FFFF8100,FIXEDVECT/0FFFFFFD0
-library=LoadModule.lib -output=LoadModule.abs UserProgram.obj resetprg.obj
intprg.obj vecttbl.obj dbsct.obj lowsrc.obj lowlvl.obj sbrk.obj
rlink -output=LoadModule.sty -form=stype -output=LoadModule.mot LoadModule.abs
```

[Reference]
An example of a command string for which the RXv2 instruction-set architecture was selected is shown.

```
lbgrx -isa=rxv2 -output=LoadModule.lib
ccrx -isa=rxv2 -output=obj UserProgram.c
ccrx -isa=rxv2 -output=obj resetprg.c
ccrx -isa=rxv2 -output=obj intprg.c
ccrx -isa=rxv2 -output=obj vecttbl.c
ccrx -isa=rxv2 -output=obj dbsct.c
ccrx -isa=rxv2 -output=obj lowsrc.c
asrx -isa=rxv2 lowlvl.src
ccrx -isa=rxv2 -output=obj sbrk.c
rlink -rom=D=R,D_1=R_1,D_2=R_2 -list=LoadModule.map
-start=B_1,R_1,B_2,R_2,B,R,SU,SI/04,PResetPRG/
0FFFF8000,C_1,C_2,C,C$DSEC,C$BSEC,C$INIT,C$VTBL,C$VECT,D_1,D_2,D,P,PInt-
PRG,W_1,W_2,W,L/0FFFF8100,EXCEPTVECT/0FFFFFF80,RESETVECT/0FFFFFFFC -=LoadMod-
ule.lib -output=LoadModule.abs UserProgram.obj resetprg.obj intprg.obj vecttbl.obj
dbsct.obj lowsrc.obj lowlvl.obj sbrk.obj
rlink -output=LoadModule.sty -form=stype -output=LoadModule.mot LoadModule.abs
```

8.5 Usage of PIC/PID Function

This section gives an overview of the PIC/PID function and describes how to create startup programs when using the PIC/PID function.

The PIC/PID function enables the code and data in the ROM to be reallocated to desired addresses without re-linkage even when the allocation addresses have been determined through previously completed linkage.

PIC stands for position independent code, and PID stands for position independent data. The PIC function generates PIC and the PID function generates PID; here, these functions are collectively called the PIC/PID function.

8.5.1 Terms Used in this Section

(1) Master and Application
In the PIC/PID function, a program whose code or data in the ROM has been converted into PIC or PID is called an application, and the program necessary to execute an application is called the master.
The master executes the application initiation processing, and also provides the shared libraries called from applications and RAM areas for applications. PIC and PID are included only in applications; the master does not have them.

(2) Shared Library
A group of functions in the master, which can be called from multiple applications.

(3) Jump Table
A program through which applications can call shared libraries.

8.5.2 Function of Each Option

The following describes the options related to the PIC/PID function.
For details of each option function, refer to the respective option description of the COMMAND REFERENCE chapter.

(1) Application Code Generation (pic and pid Options)
When the **pic** option is specified for compilation, the PIC function is enabled and the code in the code area (**P** section) becomes PIC. The PIC always uses PC relative mode to acquire branch destination addresses or function addresses, so it can be reallocated to any desired addresses even after linkage.
When the **pid** option is specified for compilation, the PID function is enabled and the data in ROM data areas (**C_8**, **C**, **C_2**, **C_1**, **W**, **W_2**, **W_1**, and **L** sections) becomes PID. A program executes relative access to the PID by using the register (PID register) that indicates the start address of the PID. The user can move the PID to any desired addresses by modifying the PID register value even after linkage.
Note that the PIC function (**pic** option) and PID function (**pid** option) are designed to operate independently. However, it is recommended to enable both functions and allocate the PIC and PID to adjacent areas. Support for independently using either the PIC or PID function and for debugging of applications where the distance between the PIC and PID is variable may or may not be available, depending on the version of the debugger. The examples described later assume that both PIC and PID functions are enabled together.

(2) Shared Library Support (jump_entries_for_pic and nouse_pid_register Options)
These options provide a function for calling the libraries of the master from an application.
The **nouse_pid_register** option should be used for master compilation to generate a code that does not use the PID register.
When the **jump_entries_for_pic** option is specified in the optimizing linkage editor at master linkage, a jump table is created to be used to call library functions at fixed addresses from an application.

(3) Sharing of RAM Area (Fsymbol Option)
This option enables variables in the master to be read or written from an application whose linkage unit differs from that of the master.
When the **Fsymbol** option is specified in the optimizing linkage editor at master linkage, a symbol table is created to be used to refer to variables at fixed addresses from an application.

8.5.3 Restrictions on Applications

(1) RAM Areas
The PID function cannot be applied to the RAM area.

(2) Simultaneous Execution of Applications
When the PIC/PID function is used, multiple copies of a single application can be stored in the ROM and each copy can be executed. However, copies of a single application cannot be executed at the same time because the RAM areas for them overlap each other.

(3) Startup
The standard startup program (created by the integrated development environment as described in section 8.3 Startup Program Creation) cannot be used to start up an application without change. Create a startup program as described in 8.5.7 Application Startup.

8.5.4 System Dependent Processing Necessary for PIC/PID Function

The following processing should be prepared by the user depending on the system specifications.

(1) Initialization of Master
Execute the same processing as that for a usual program which does not use the PIC/PID function.

(2) Initiation of Application from the Master
Set the PID register to the start address of the application PID and branch to the PIC start address to initiate the application.

(3) Initialization of Application
Initialize the section and execute the **main** function of the application.

(4) Termination of Application
After execution of the **main** function, return execution to the master.

8.5.5 Combinations of Code Generating Options

When the master and application are built, the option settings related to the PIC/PID function should be matched between the objects that compose the master and application.

The following shows the rules for specifying options for each object compilation and the conditions of option settings in other objects that can be linked.

(1) Master

When building the master, specify the PIC/PID function options as shown in Table 8.6.

Table 8.6 Rules for Specifying PIC/PID Function Options in Master

Option Name	For Compilation	Conditions on Setting the Option for Linkable Objects
pic	× Not allowed	**pic** is not specified
pid	× Not allowed	**pid** is not specified
nouse_pid_register	O Can be specified except the standard library and setting PID register of the startup program	No conditions
fint_register	O Can be specified	**fint_register** with the same parameters must be specified
base	O Can be specified	**base** with the same parameters must be specified

(2) Application

When building an application, specify the PIC/PID function options as shown in Table 8.7.

Table 8.7 Rules for Specifying PIC/PID Function Options in Application

Option Name	For Compilation	Conditions on Setting the Option for Linkable Objects
pic	O Can be specified	**pic** is necessary
pid	O Can be specified	**pid** is necessary
nouse_pid_register	× Not allowed	**nouse_pid_register** is not specified
fint_register	O Can be specified	**fint_register** with the same parameters must be specified
base	O : Can be specified	**base*** with the same parameters must be specified

Note * When **pid** is specified, **base=rom=<register>** is not allowed.

(3) Between Master and Application

In the master and application, the PIC/PID function options should be specified as shown in Table 8.8.

Table 8.8 Rules for Combinations of PIC/PID Function Options between Master and Application

Options in Application	Options in Master
pic	No conditions
pid	**nouse_pid_register** is necessary if application calls functions of master
fint_register	**fint_register** with the same parameters is necessary
base	**base*** with the same parameters is necessary

Note * When **pid** is specified, **base=rom=<register>** is not allowed.

8.5.6 Master Startup

The processing necessary to start up the master is the same as that for a usual program that does not use the PIC/PID function except for the two processes described below. Add these two processes to the startup processing created according to section 7.3, Startup Program Creation.

(1) Initiation of and Return from Application
Set up the PID register in the **main** function and branch to the PIC entry address to initiate the application. In addition, a means for returning from the application to the master should be provided.

(2) Reference to Shared Library Functions to be Used
The shared libraries to be used by the application should be referred to also by the master in advance.
The following shows an example for calling a PIC/PID application from the **main** function.
This example assumes the following conditions:

- After application execution, control can be returned to the master through the RTS instruction.
- The application does not pass a return value.
- The PID initiation address (PIC_entry) and PID start address (PID_address) for the application are known and fixed when the master is built.
- R13 is used as the PID register.
- Initialization of the section areas on the application side is not done on the master side.
- The application uses only the **printf** function as the shared library.

Example:

```
/* Master-Side Program */
/* Initiates the PIC/PID application. */
/* (For the system that the application does not pass */
/*  a return value and execution returns through RTS) */
#include <stdio.h>
#pragma inline_asm Launch_PICPID_Application
void Launch_PICPID_Application(void *pic_entry, void *pid_address)
{
    MOV.L     R2,__PID_R13
    JSR       R1
}
int main()
{
    void *PIC_entry   = (void*)0x500000;  /* PIC initiation address */
    void *PID_address = (void*)0x120000;  /* PID start address */

    /* (1) Initiation of and Return from Application */
    Launch_PICPID_Application(PIC_entry, PID_address);

    return 0;
}

/* (2) Reference to Shared Library Functions to be Used */
void *_dummy_ptr = (void*)printf;  /* printf function */
```

8.5.7 Application Startup

Specify the following in the application.
The items marked with **[Optional]** may be unnecessary in some cases.

(1) Preparation of Entry Point (PIC Initiation Address)
This is the address from which the application is initiated.

(2) Initialization of Stack Pointer [Optional]
This processing is not necessary when the application shares the stack with the master.
When necessary, add appropriate settings by referring to section 7.3.2 (2).

(3) Initialization of General Registers Used as Base Registers [Optional]
This processing is not necessary when no base register is used.
When necessary, add appropriate settings by referring to section 7.3.2 (3).

(4) Initialization Processing of Sections [Optional]
This processing is not necessary when the master initializes them.
When necessary, add appropriate settings by referring to the example shown later.
Note that the processing described in section 7.3.2 (5) cannot be used without change.

(5) Initialization Processing of Libraries [Optional]
This processing is not necessary when no standard library is used.
When necessary, add appropriate settings by referring to section 7.3.2 (6).
Initialization of PSW for main Function Execution [Optional]
Specify interrupt masks or move to the user mode as necessary.
Add appropriate settings by referring to sections 7.3.2 (8) and 7.3.2 (9).

(6) User Program Execution
Execute the **main** function.
Specify the processing by referring to section 7.3.2 (10).

The following shows an example of application startup.
The processing is divided into three files.

- **startup_picpid.c**: Body of the startup processing.
- **initsct_pid.src**: Section initialization for PID; **_INITSCT_PID**.
  This is created by modifying the **_INITSCT** function described in section 7.3.2 (5) to support the PID function.
  "__PID_REG" in the program will be converted into PID register when the assembling.
- **initiolib.c**; Contains **_INITLIB**, which initializes the standard libraries.
  This is created by modifying the code described in section 7.3.2 (6) to be used for the application.
  [startup_picpid.c]

```
// Initialization Processing Described in Section 7.3.2(5)
#pragma section C C$DSEC //Section name is set to C$DSEC
const struct {
    void *rom_s; //Start address member of the initialized data section in ROM
    void *rom_e; //End address member of the initialized data section in ROM
    void *ram_s; //Start address member of the initialized data section in RAM
} DTBL[] = {__sectop("D"), __secend("D"), __sectop("R")};
#pragma section C C$BSEC //Section name is set to C$BSEC
const struct {
    void *b_s; //Start address member of the uninitialized data section
    void *b_e; //End address member of the uninitialized data section
} BTBL[] = {__sectop("B"), __secend("B")};

extern void main(void);
extern void _INITLIB(void); // Library initialization processing described
                            //in section 7.3.2 (6)
#pragma entry application_pic_entry
void application_pic_entry(void)
{
    _INITSCT_PICPID();
    _INITLIB();
     main();
}
```

[initsct_pid.src]

```
; Section Initialization Routine for PID Support
; ** Note ** Check the PID register.
; This code assumes that R13 is used as the PID register. If another
; register is used as the PID register, modify the description related to R13
; in the following code to the register assigned as the PID register
; in your system.
    .glb   __INITSCT_PICPID
    .glb   __PID_TOP
    .section  C$BSEC,ROMDATA,ALIGN=4
    .section  C$DSEC,ROMDATA,ALIGN=4
    .section  P,CODE

__INITSCT_PICPID:               ; function: _INITSCT
    .STACK    __INITSCT_PICPID=28
    PUSHM     R1-R6
    ADD       #-__PID_TOP,__PID_REG,R6 ;   How long distance PID moves
;;;
;;; clear BBS(B)
;;;
    ADD       #TOPOF C$BSEC,  R6, R4
    ADD       #SIZEOF C$BSEC, R4, R5
    MOV.L     #0, R2
    BRA       next_loop1

loop1:
    MOV.L     [R4+], R1
    MOV.L     [R4+], R3
    CMP       R1, R3
    BLEU      next_loop1
    SUB       R1, R3
    SSTR.B
next_loop1:
    CMP       R4,R5
    BGTU      loop1

;;;
;;; copy DATA from ROM(D) to RAM(R)
;;;
    ADD       #TOPOF C$DSEC,  R6, R4
    ADD       #SIZEOF C$DSEC, R4, R5
    BRA       next_loop3

loop3:
    MOV.L     [R4+], R2
    MOV.L     [R4+], R3
    MOV.L     [R4+], R1
    CMP       R2, R3
    BLEU      next_loop3
    SUB       R2, R3
    ADD       R6, R2       ; Adjust for real address of PID
    SMOVF
next_loop3:
    CMP       R4, R5
    BGTU      loop3
    POPM      R1-R6
    RTS

    .end
```

[initiolib.c]

```
#include <stdio.h>
#include <stdlib.h>
#define IOSTREAM 3
const size_t _sbrk_size = 520; // Specifies the minimum unit of the heap area
                               // allocation size. (Default: 1024)

void _INIT_LOWLEVEL(void);
void _INIT_OTHERLIB(void);

void _INITLIB (void)
{
    _INIT_LOWLEVEL(); // Initial settings for low-level interface routines
    _INIT_IOLIB();    // Initial settings for I/O library
    _INIT_OTHERLIB(); // Initial settings for rand and strtok functions
}
void _INIT_LOWLEVEL(void)
{ // Make necessary settings for low-level library
}
void _INIT_OTHERLIB(void)
{
     srand(1); // Initial settings necessary when the rand function is used
}
```

9. FUNCTION CALL INTERFACE SPECIFICATIONS

9.1 Function Calling Interface

This chapter describes how to handle, for example, arguments when calling functions written in the C or C++ language from a program when using the **CC-RX** compiler.

The compiler generates code in accord with the following descriptions.

Follow the rules described in this chapter when creating functions which interface the C/C++ language with assembler code.

With regard to interrupt functions, also refer to section 4.2.4 Using Extended Specifications (3) Interrupt Function Creation.

9.1.1 Rules Concerning the Stack

(1) Stack Pointer

Valid data must not be stored in a stack area with an address lower than the stack pointer (in the direction of address H'0), since the data may be destroyed by an interrupt process.

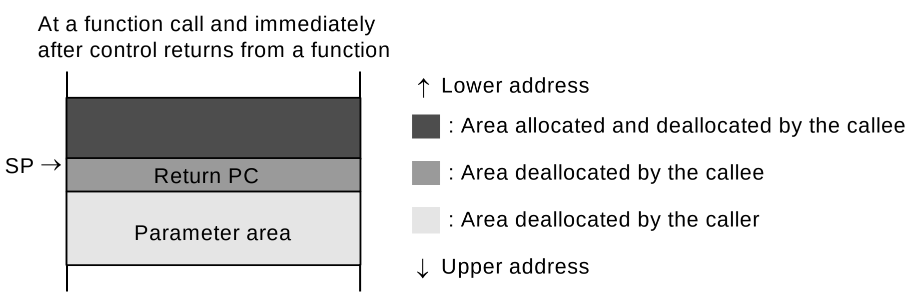
(2) Allocating and Deallocating Stack Frames

In a function call (immediately after the **JSR** or the **BSR** instruction has been executed), the stack pointer indicates the lowest address of the stack used by the calling function. Allocating and setting data at addresses greater than this address must be done by the caller.

After the callee deallocates the area it has set with data, control returns to the caller usually with the **RTS** instruction. The caller then deallocates the area having a higher address (the return value address and the parameter area).

Figure 9.1 illustrates the stack frame status immediately after a function call.

Figure 9.1 Allocation and Deallocation of a Stack Frame



9.1.2 Rules Concerning Registers

Registers having the same value before and after a function call is not guaranteed for some registers; some registers may change during a function call. Some registers are used for specific purposes according to the option settings. Table 9.1 shows the rules for using registers.

Table 9.1 Rules to Use Registers

Register	Register Value Does Not Change During Function Call	Function Entry	Function Exit	High-Speed Interrupt Register*1	Base Register*2	PID Register*3
R0	Guaranteed	Stack pointer	Stack pointer	—	—	—
R1	Not guaranteed	Parameter 1	Return value 1	—	—	—
R2	Not guaranteed	Parameter 2	Return value 2	—	—	—
R3	Not guaranteed	Parameter 3	Return value 3	—	—	—
R4	Not guaranteed	Parameter 4	Return value 4	—	—	—

Register	Register Value Does Not Change During Function Call	Function Entry	Function Exit	High-Speed Interrupt Register*1	Base Register*2	PID Register*3
R5	Not guaranteed	—	(Undefined)	—	—	—
R6	Guaranteed	—	(Value at function entry is held)	—	—	—
R7	Guaranteed	—	(Value at function entry is held)	—	—	—
R8	Guaranteed	—	(Value at function entry is held)	—	O	—
R9	Guaranteed	—	(Value at function entry is held)	—	O	O
R10	Guaranteed	—	(Value at function entry is held)	O	O	O
R11	Guaranteed	—	(Value at function entry is held)	O	O	O
R12	Guaranteed	—	(Value at function entry is held)	O	O	O
R13	Guaranteed	—	(Value at function entry is held)	O	O	O
R14	Not guaranteed	—	(Undefined)	—	—	—
R15	Not guaranteed	Pointer to return value of structure	(Undefined)	—	—	—
DR0 to DR15 [V3.01.00 or later]	Guaranteed	—	(Value at function entry is held)	—	—	—
DCMR [V3.01.00 or later]	Guaranteed	—	(Value at function entry is held)	—	—	—
DPSW DECNT DEPC [V3.01.00 or later]	Not guaranteed	—	(Undefined)	—	—	—
ISP USP	Same as R0 when used as the stack pointer. In other cases, the values do not change. *4			—	—	—
PC	—	Program counter*5		—	—	—
PSW	Not guaranteed	—	(Undefined)	—	—	—
FPSW	Not guaranteed	—	(Undefined)	—	—	—
ACC	Not guaranteed*6	—	(Undefined) *6	—	—	—
INTB BPC BPSW FINTV	—	No change*4	—	—	—	—

Notes 1. The high-speed interrupt function may use some or all four registers among R10 to R13, depending on the **fint_register** option. Registers assigned to the high-speed interrupt function cannot be used for other purposes. For details on the function, refer to the description on the option.

Notes 2. The base register function may use some or all six registers among R8 to R13, depending on the **base** option. Registers assigned to the base register function cannot be used for other purposes. For details on the function, refer to the description on the option.

Notes 3. The PID function may use one of R9 to R13, depending on the **pid** option. The register assigned to the PID function cannot be used for other purposes. For details on the function, refer to the description on the option.

Notes 4. This does not apply in the case when the registers are set or modified through an intrinsic function or **#pragma inline_asm**.

Notes 5. This depends on the specifications of the instruction used for function calls. To call a function, use BSR, JSR, BRA, or JMP.

Notes 6. For the instructions that modify the ACC (accumulator), refer to the software manual for the target RX series product.

9.1.3 Rules Concerning Setting and Referencing Parameters

General rules concerning parameters and the method for allocating parameters are described.
Refer to section 9.1.5 Examples of Parameter Allocation, for details on how to actually allocate parameters.

(1) Passing Parameters
A function is called after parameters have been copied to a parameter area in registers or on the stack. Since the caller does not reference the parameter area after control returns to it, the caller is not affected even if the callee modifies the parameters.

(2) Rules on Type Conversion

(a) Parameters whose types are declared by a prototype declaration are converted to the declared types.

(b) Parameters whose types are not declared by a prototype declaration are converted according to the following rules.

- **int** type of 2 bytes or less is converted to a 4-byte **int** type.
- **float** type parameters are converted to **double** type parameters.
- Types other than the above are not converted.

Example

```
void p(int,... );
void f( )
{
  char c;
  p(1.0, c);
}
```

c is converted to a 4-byte **int** type because a type is not declared for the parameter.

1.0 is converted to a 4-byte **int** type because the type of the parameter is **int**.

(3) Parameter Area Allocation
Parameters are allocated to registers or to a parameter area on the stack. Figure 9.2 shows the parameter-allocated areas.
Following the order of their declaration in the source program, parameters are normally allocated to the registers starting with the smallest numbered register. After parameters have been allocated to all registers, parameters are allocated to the stack. However, in some cases, such as a function with variable-number parameters, parameters are allocated to the stack even though there are empty registers left. The **this** pointer to a nonstatic function member in a C++ program is always assigned to R1.
Table 9.2 lists general rules on parameter area allocation.

Figure 9.2 Parameter Area Allocation

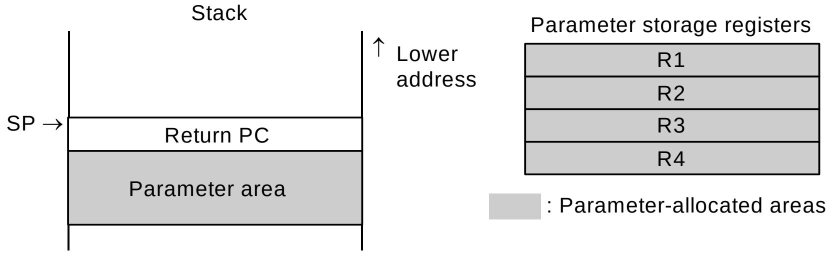


Table 9.2 General Rules on Parameter Area Allocation

Parameters Allocated to Registers			Parameters Allocated to Stack
Target Type	Parameter Storage Registers	Allocation Method	
signed char, (unsigned) char, bool, _Bool, (signed) short, unsigned short, (signed) int, unsigned int, (signed) long, unsigned long, float, double*1, long double*1, pointer, pointer to a data member, and reference	One register among R1 to R4	Sign extension is performed for **signed char** or **(signed) short** type, and zero extension is performed for **(unsigned) char** type, and the results are allocated. All other types are allocated without any extension performed.	(1) Parameters whose types are other than target types for register passing (2) Parameters of a function which has been declared by a prototype declaration to have variable-number parameters*3 (3) When the number of registers not yet allocated with parameters among R1 to R4 is smaller than the number of registers needed to allocate parameters
(signed) long long, unsigned long long, double*2, and long double*2	Two registers among R1 to R4	The lower four bytes are allocated to the smaller numbered register and the upper four bytes are allocated to the larger numbered register.	
Structure, union, or class whose size is a multiple of 4 not greater than 16 bytes	Among R1 to R4, a number of registers obtained by dividing the size by 4	From the beginning of the memory image, parameters are allocated in 4-byte units to the registers starting with the smallest numbered register.	

Notes 1. When **dbl_size=8** is not specified.

Notes 2. When **dbl_size=8** is specified.

Notes 3. If a function has been declared to have variable parameters by a prototype declaration, parameters which do not have a corresponding type in the declaration and the immediately preceding parameter are allocated to the stack. For parameters which do not have a corresponding type, an integer of 2 bytes or less is converted to **long** type and **float** type is converted to **double** type so that all parameters will be handled with a boundary alignment number of 4.

Example

```
int f2(int,int,int,int,...);
:
f2(a,b,c,x,y,z);  ? x, y, and z are allocated to the stack.
```

(4) Allocation Method for Parameters Allocated to the Stack
The address and allocation method to the stack for the parameters that are shown in Table 9.2 as parameters allocated to the stack are as follows:

- Each parameter is placed at an address matching its boundary alignment number.

- Parameters are stored in the parameter area on the stack in a manner so that the leftmost parameter in the parameter sequence will be located at the deep end of the stack. To be more specific, when parameter **A** and its right-hand parameter **B** are both allocated to the stack, the address of parameter **B** is calculated by adding the occupation size of parameter **A** to the address of parameter **A** and then aligning the address to the boundary alignment number of parameter **B**.

9.1.4 Rules Concerning Setting and Referencing Return Values

General rules concerning return values and the areas for setting return values are described.

(1) Type Conversion of a Return Value
A return value is converted to the data type returned by the function.

Example

```
 long f( );
long f( )
{
     float x;
    return x;     <---- The return value is converted to long type
                        by a prototype declaration
}
```

(2) Return Value Setting Area
The return value of a function is written to either a register or memory depending on its type. Refer to Table 9.3 for the relationship between the type and the setting area of the return value.

Table 9.3 Return Value Type and Setting Area

Return Value Type	Return Value Setting Area
signed char, (unsigned) char, (signed) short, unsigned short, (signed) int, unsigned int, (signed) long, unsigned long, float, double*2, long double*2, pointer, bool, _Bool, reference, and pointer to a data member	R1 Note however that the result of sign extension is set for **signed char** or **(signed) short** type, and the result of zero extension is set for **(unsigned) char** or **unsigned short** type.
double*3, long double*3, (signed) long long, and unsigned long long	R1, R2 The lower four bytes are set to R1 and the upper four bytes are set to R2.
Structure, union, or class whose size is 16 bytes or less and is also a multiple of 4	They are set from the beginning of the memory image in 4-byte units in the order of R1, R2, R3, and R4.
Structure, union, or class other than those above	Return value setting area (memory)*1

Notes 1. When a function return value is to be written to memory, the return value is written to the area indicated by the return value address. The caller must allocate the return value setting area in addition to the parameter area, and must set the address of the return value setting area in R15 before calling the function.

Notes 2. When **dbl_size=8** is not specified.

Notes 3. When **dbl_size=8** is specified.

9.1.5 Examples of Parameter Allocation

Examples of parameter allocation are shown in the following. Note that addresses increase from the right side to the left side in all figures (upper address is on the left side).

Examples 1. Parameters matching the type to be passed to registers are allocated, in the order in which they are declared, to registers R1 to R4.
If there is a parameter that will not be allocated to registers midway, parameters after that will be allocated to registers. The parameter will be placed on the stack at an address corrected to match the boundary alignment number of that parameter.

```
int f(
    unsigned char,
    long long,
    long long,
    short,
    int,
    char,
    short,
    char,
    char,
    short);
        :
f(1,2,3,4,5,6,7,8,9,10);
/*
** 1, 2, and 4 are allocated to registers
*/
```

<Registers>

R1	0x000000 (Zero extension)		0x01
R2	0x00000002		
R3	0x00000000		
R4	0x0000 (Sign extension)	0x0004	

<Stack>

*(R0+0)	0x00000003		
*(R0+4)	0x00000000		
*(R0+8)	0x00000005		
*(R0+12)	0x0007	Empty area	0x06
*(R0+16)	0x000A	0x09	0x08

Examples 2. Parameters of a structure or union whose size is 16 bytes or less and is also a multiple of 4 are allocated to registers. Parameters of all other structures and unions are allocated to the stack.

```
union U {int a[2]; int b;} u;
struct S {short d; char c[4];} s;
struct T {char g; char f[2]; char e;} t;
int f(union U, struct S, struct T);
:
f(u, s, t);
/*
** u is allocated to a register because it is 8 bytes
** s is allocated to the stack because it is 6 bytes
** t is allocated to a register because it is 4 bytes
*/
```

<Registers>

R1	u.a[0] (=u.b)			
R2	u.a[1]			
R3	t.e	t.f[1]	t.f[0]	t.g

<Stack>

*(R0+0)	s.c[1]	s.c[0]	s.d	
*(R0+4)	Empty area		s.c[3]	s.c[2]

Examples 3. When declared in a prototype declaration as a function with a variable-number of parameters, the parameters without corresponding types and the immediately preceding parameter are allocated to the stack in the order in which they are declared.

```
int f(int, float, int, int, ...)
    :
f(0, 1.0, 2, 3, 4)
```

<Registers>

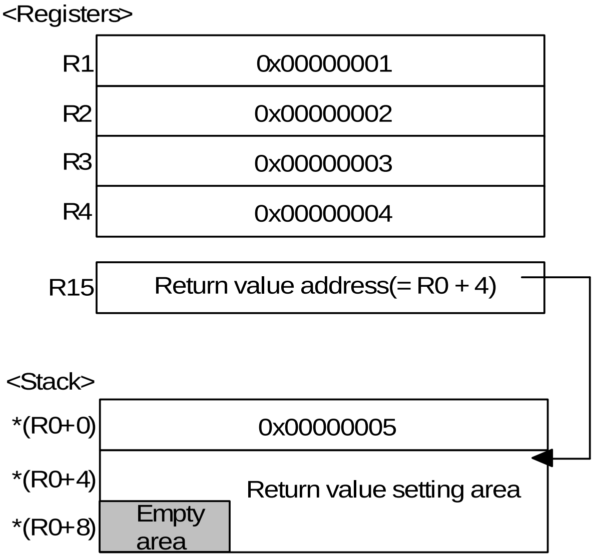
R1	0x00000000
R2	0x3F800000
R3	0x00000002

<Stack>

*(R0+0)	0x00000003
*(R0+4)	0x00000004

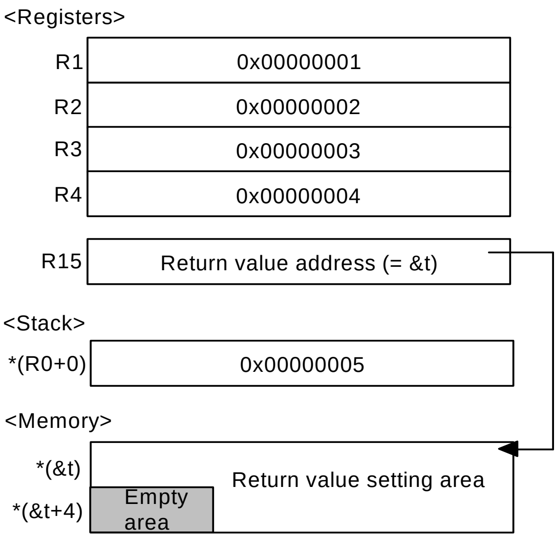
Examples 4. When the type returned by a function is more than 16 bytes, or for a structure or union that is not the size of a multiple of 4, the return value address is set to R15.

```
struct S{char a[7];};
struct S f(
   int a1,
   int a2,
   int a3,
   int a4,
   int a5);
   :
f(1,2,3,4,5);
```



Examples 5. When setting the return value to memory, normally a stack is allocated, as shown in example 4. In the case of setting the return value to a variable, however, no stack is allocated and it is directly set to the memory area for that variable. In this case, the address for the variable is set to R15.

```
struct S{char a[7];}t;
struct S f(
   int a1,
   int a2,
   int a3,
   int a4,
   int a5);
   :
t=f(1,2,3,4,5);
```



9.2 Method for Mutual Referencing of External Names between Compiler and Assembler

This section describes mutual referencing between the compiler and assembler.

External names which have been declared in a C/C++ program can be referenced and updated in both directions between the C/C++ program and an assembly-language program. The compiler treats the following items as external names.

- Global variables which are not declared as **static** storage classes (C/C++ programs)
- Variable names declared as **extern** storage classes (C/C++ programs)
- Function names not specified as **static** storage classes (C programs)
- Non-member, non-inline function names not specified as **static** storage classes (C++ programs)
- Non-inline member function names (C++ programs)
- Static data member names (C++ programs)

9.2.1 Referencing Assembly-Language Program External Names in C/C++ Programs

In assembly-language programs, **.GLB** is used to declare external symbol names (preceded by an underscore (_)).
In C/C++ programs, symbol names (not preceded by an underscore) are declared using the **extern** keyword.
[Example of assembly-language source]

```
     .glb _a, _b
     .SECTION D,ROMDATA,ALIGN=4
_a:  .LWORD 1
_b:  .LWORD 1
     .END
```

[Example of C source]

```
extern int a,b;
void f()
{
     a+=b;
}
```

9.2.2 Referencing C/C++ Program External Names (Variables and C Functions) from Assembly-Language Programs

A C/C++ program can define external variable names (without an underscore (_)).
In an assembly-language program, **.GLB** is used to declare an external name (preceded by an underscore).
[Example of C source]

```
int a;
```

[Example of assembly-language source]

```
     .GLB _a
     .SECTION P,CODE
     MOV.L #A_a,R1
     MOV.L [R1],R2
     ADD #1,R2
     MOV.L R2,[R1]
     RTS
     .SECTION D,ROMDATA,ALIGN=4
A_a: .LWORD _a
     .END
```

9.2.3 Referencing C++ Program External Names (Functions) from Assembly-Language Programs

By declaring functions to be referenced from an assembly-language program using the **extern "C"** keyword, the function can be referenced using the same rules as in (2) above. However, functions declared using **extern "C"** cannot be overloaded.

[Example of C++ source]

```
extern "C"
void sub()
{
     :
}
```

[Example of assembly-language source]

```
.GLB _sub
.SECTION P,CODE
  :
PUSH.L R13
MOV.L 4[R0],R1
MOV.L R3,R12
MOV.L #_sub,R14
JSR R14
POP R13
RTS
  :
.END
```

10. MESSAGES

10.1 GENERAL

This document describes internal error message, error message, fatal error message, information message, warning message and MISRA-C detection message that Renesas Tool outputs.

10.2 MESSAGE FORMATS

(1) When the file name and line number are included

```
file-name (line-number) : message-type component-number message-number : message
```

(2) When the file name and line number aren't included

```
message-type component-number message-number : message
```

Remark Following contents are output as the continued character string.
MESSAGE TYPES : 1 alphabetic character
COMPONENT NUMBERS : 05
MESSAGE NUMBERS : 5 digits

10.3 MESSAGE TYPES

Table 10.1 Message Type (CC-RX (V2.00.00 or higher))

Message Type	Description
C	Internal error : Processing is aborted. No output objects are generated.
E	Error : Processingn is aborted if a set number of errors occur. No output objects are generated.
F	Fatal error : Processing is aaborted. No output objects are generated.
M	Information : Informational message. Check the message and continue the process.
W	Warning : Processing continues. Output objects are generated (They might not be what the user intended).

10.4 MESSAGE NUMBERS

The message numbers of the CC-RX (V2.00.00 or higher) are 5 digits number output following component number (05).

10.5 MESSAGES

This chapter describes the messages displayed by Renesas Tool.

10.5.1 Internal Errors

Table 10.2 Internal Errors

C0510000	[Message]	Internal error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0511200	[Message]	Internal error(*error-information*).
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0519996	[Message]	Out of memory.
	[Explanation]	The amount of data input (source file name and specified options) to the ccrx command is too large.
	[Action by User]	Divide the data input to the ccrx command, and then perform startup several times.
C0519997	[Message]	Internal error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0520000	[Message]	Internal Error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0529000	[Message]	Internal Error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0530001	[Message]	Internal Error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0530002	[Message]	Internal Error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0530003	[Message]	Internal Error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0530004	[Message]	Internal Error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0530005	[Message]	Internal Error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0530006	[Message]	Internal Error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0554098	[Message]	Internal Error.
	[Explanation]	An internal error occurred during processing by the assembler.
	[Action by User]	Make a note of the internal error number, file name, line number, and comment in the message, and contact the support department of the vendor.
C0564000	[Message]	Internal error : ("*internal error number*") "*file line number*" / "*comment*"
	[Explanation]	An internal error occurred during processing by the linker.
	[Action by User]	Make a note of the internal error number, file name, line number, and comment in the message, and contact the support department of the vendor.

C0564001	[Message]	Internal Error.
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.
C0590001	[Message]	Internal error
	[Action by User]	Please contact your vendor or your Renesas Electronics overseas representative.

10.5.2 Errors

Table 10.3 Errors

E0511101	[Message]	"*path*" specified by the "*character string*" option is a folder. Specify an input file.
E0511102	[Message]	The file "*file*" specified by the "*character string*" option is not found.
E0511103	[Message]	"*path*" specified by the "*character string*" option is a folder. Specify an output file.
E0511104	[Message]	The output folder "*folder*" specified by the "*character string*" option is not found.
E0511107	[Message]	"*path*" specified by the "*character string*" option is not found.
	[Explanation]	"*path*" (*file-name or folder*) specified in the "*character string*" option was not found.
E0511108	[Message]	The "*character string*" option is not recognized.
E0511109	[Message]	The "*character string*" option can not have an argument.
E0511110	[Message]	The "*character string*" option requires an argument.
	[Explanation]	The "*character string*" option requires an argument. Specify the argument.
E0511111	[Message]	The "*character string*" option can not have a parameter.
E0511112	[Message]	The "*character string*" option requires a parameter.
	[Explanation]	The "*character string*" option requires a parameter. Specify the parameter.
E0511113	[Message]	Invalid argument for the "*character string*" option.
E0511117	[Message]	Invalid parameter for the "*character string*" option.
E0511118	[Message]	Symbol is required for the "*character string*" option.
E0511120	[Message]	Specify a value (*value1* - *value2*) for the "*character string*" option.
	[Explanation]	The value of the specified size option is outside the range of minimum value to maximum value.
	[Action by User]	Specify a size option value between the minimum and maximum values.
E0511122	[Message]	The argument for the "*character string*" option must be an object file.
E0511127	[Message]	The specified device is not supported.
E0511129	[Message]	Command file "*file*" is read more than once.
E0511130	[Message]	Command file "*file*" cannot be read.
E0511131	[Message]	Syntax error in command file "*file*".
E0511132	[Message]	Failed to create temporary folder.
E0511133	[Message]	The parameter for the "*character string*" option must be a folder when multiple source files are specified.
E0511134	[Message]	Input file "*file*" is not found.
E0511135	[Message]	"*path*" specified as an input file is a folder.
E0511136	[Message]	Failed to delete a temporary file "*file*".
E0511137	[Message]	Failed to delete a temporary folder "*folder*".
E0511138	[Message]	Failed to open an input file "*file*".
E0511139	[Message]	Failed to open an output file "*file*".
E0511140	[Message]	Failed to close an input file "*file*".

E0511141	[Message]	Failed to write an output file "*file*".
E0511142	[Message]	Multiple source files are not allowed when the "*character string*" option is specified.
E0511145	[Message]	"*character string2*" specified in the "*character string1*" option is not available.
E0511148	[Message]	"*file name*" is specified as an output file for the different options.
	[Action by User]	"*file name*" is specified as an output file for the different options. Specify a different file name.
E0511150	[Message]	The "*character string1*" option and the "*character string2*" option are inconsistent.
E0511152	[Message]	The "*character string1*" option needs the "*character string2"* option.
E0511154	[Message]	Component file "*file name*" for the CC-RX is not found. Reinstall the CC-RX.
	[Explanation]	The component file "*file name*" for the CC-RX is not found. Reinstall the CC-RX.
E0511155	[Message]	The "*character string*" option needs other option(s).
E0511157	[Message]	The "*character string1*" option or the "*character string2*" option must be specified for this device.
E0511158	[Message]	The "*character string*" option is not supported for this device.
E0511159	[Message]	When the "*character string*" option is specified, source files cannot be input.
E0511160	[Message]	The "*character string*" option must be specified for this device.
E0511161	[Message]	Failed to delete a file "*file*".
E0511167	[Message]	Illegal section naming.
	[Explanation]	There is an error in section naming. The same section name is specified for different use of the section.
E0511173	[Message]	Failed to access a temporary file
E0511175	[Message]	Neither isa nor cpu is specified.
E0511176	[Message]	Both "-isa" option and "-cpu" option are specified.
E0511178	[Message]	"*character string*" option is unavailable because the license of *version* Professional edition is not found. Please consider purchasing the product of Professional edition.
E0511200	[Message]	Internal error(*error-information*).
E0512001	[Message]	Failed to delete a temporary file "*file*".
E0520001	[Message]	Last line of file ends without a newline.
	[Action by User]	The last line in the file does not end with a line break. Add a line break.
E0520002	[Message]	Last line of file ends with a backslash.
	[Explanation]	There is a backslash at the end of the last line of the file. Delete it.
E0520005	[Message]	Could not open source file "*file name*".
E0520006	[Message]	Comment unclosed at end of file.
	[Action by User]	There is an unclosed comment at the end of the file. Make sure that there are no unclosed comments.
E0520007	[Message]	Unrecognized token.
	[Action by User]	Unknown token. Check the indicated location.

E0520008	[Message]	Missing closing quote.
	[Action by User]	The string is missing a closing quotation mark. Make sure that there are no unclosed quotation mark.
E0520010	[Message]	"#" not expected here.
	[Explanation]	There is a "#" character in an invalid location.
E0520011	[Message]	Unrecognized preprocessing directive.
E0520012	[Message]	Parsing restarts here after previous syntax error.
E0520013	[Message]	Expected a file name.
E0520014	[Message]	Extra text after expected end of preprocessing directive.
E0520017	[Message]	Expected a "]".
E0520018	[Message]	Expected a ")".
E0520019	[Message]	Extra text after expected end of number.
E0520020	[Message]	Identifier "*character string*" is undefined.
E0520022	[Message]	Invalid hexadecimal number.
E0520023	[Message]	Integer constant is too large.
E0520024	[Message]	Invalid octal digit.
	[Explanation]	Invalid hexadecimal number. Hexadecimal numbers cannot contain '8' or '9'.
E0520025	[Message]	Quoted string should contain at least one character.
E0520026	[Message]	Too many characters in character constant.
E0520027	[Message]	Character value is out of range.
E0520028	[Message]	Expression must have a constant value.
E0520029	[Message]	Expected an expression.
E0520030	[Message]	Floating constant is out of range.
E0520031	[Message]	Expression must have integral type.
E0520032	[Message]	Expression must have arithmetic type.
E0520033	[Message]	Expected a line number
	[Explanation]	The line number after the "#line" statement does not exist.
E0520034	[Message]	Invalid line number
	[Explanation]	The line number after the "#line" statement is invalid.
E0520036	[Message]	The #if for this directive is missing.
E0520037	[Message]	The #endif for this directive is missing.
E0520038	[Message]	Directive is not allowed -- an #else has already appeared.
	[Explanation]	This directive is invalid because there is already an "#else" statement.
E0520039	[Message]	Division by zero.
E0520040	[Message]	Expected an identifier.
E0520041	[Message]	Expression must have arithmetic or pointer type.
E0520042	[Message]	Operand types are incompatible ("*type1*" and "*type2*").

E0520044	[Message]	Expression must have pointer type.
E0520045	[Message]	#undef may not be used on this predefined name.
E0520046	[Message]	"*macro*" is predefined; attempted redefinition ignored.
	[Explanation]	The macro "*macro*" is predefined. It cannot be redefined.
E0520047	[Message]	Incompatible redefinition of macro "*macro*" (declared at line *number*).
	[Explanation]	The redefinition of macro "*macro*" is not compatible with the definition at line number.
E0520049	[Message]	Duplicate macro parameter name.
E0520050	[Message]	"##" may not be first in a macro definition.
E0520051	[Message]	"##" may not be last in a macro definition.
E0520052	[Message]	Expected a macro parameter name.
E0520053	[Message]	Expected a ":".
E0520054	[Message]	Too few arguments in macro invocation.
E0520055	[Message]	Too many arguments in macro invocation.
E0520056	[Message]	Operand of sizeof may not be a function.
E0520057	[Message]	This operator is not allowed in a constant expression.
E0520058	[Message]	This operator is not allowed in a preprocessing expression.
E0520059	[Message]	Function call is not allowed in a constant expression.
E0520060	[Message]	This operator is not allowed in an integral constant expression.
E0520061	[Message]	Integer operation result is out of range.
E0520062	[Message]	Shift count is negative.
E0520063	[Message]	Shift count is too large.
E0520064	[Message]	Declaration does not declare anything.
E0520065	[Message]	Expected a ";".
E0520066	[Message]	Enumeration value is out of "int" range.
E0520067	[Message]	Expected a "}".
E0520069	[Message]	Integer conversion resulted in truncation
	[Explanation]	The conversion result of the integer type was truncated.
E0520070	[Message]	Incomplete type is not allowed.
E0520071	[Message]	Operand of sizeof may not be a bit field.
E0520075	[Message]	Operand of "*" must be a pointer.
E0520077	[Message]	This declaration has no storage class or type specifier.
E0520078	[Message]	A parameter declaration may not have an initializer.
E0520079	[Message]	Expected a type specifier.
E0520080	[Message]	A storage class may not be specified here.

E0520081	[Message]	More than one storage class may not be specified.
	[Explanation]	Multiple storage class areas have been specified. Only one storage class area can be specified.
E0520083	[Message]	Type qualifier specified more than once.
	[Explanation]	Multiple type qualifiers have been specified. It is not possible to specify more than one type qualifier.
E0520084	[Message]	Invalid combination of type specifiers.
E0520085	[Message]	Invalid storage class for a parameter.
E0520086	[Message]	Invalid storage class for a function.
E0520087	[Message]	A type specifier may not be used here.
E0520088	[Message]	Array of functions is not allowed.
E0520089	[Message]	Array of void is not allowed.
E0520090	[Message]	Function returning function is not allowed.
E0520091	[Message]	Function returning array is not allowed.
E0520092	[Message]	Identifier-list parameters may only be used in a function definition.
E0520093	[Message]	Function type may not come from a typedef.
E0520094	[Message]	The size of an array must be greater than zero.
E0520095	[Message]	Array is too large.
E0520096	[Message]	A translation unit must contain at least one declaration.
E0520097	[Message]	A function may not return a value of this type.
E0520098	[Message]	An array may not have elements of this type.
E0520099	[Message]	A declaration here must declare a parameter.
E0520100	[Message]	Duplicate parameter name.
E0520101	[Message]	"*symbol*" has already been declared in the current scope.
E0520102	[Message]	Forward declaration of enum type is nonstandard.
E0520103	[Message]	Class is too large.
E0520104	[Message]	Struct or union is too large.
E0520105	[Message]	Invalid size for bit field.
E0520106	[Message]	Invalid type for a bit field.
E0520107	[Message]	Zero-length bit field must be unnamed.
E0520109	[Message]	Expression must have (pointer-to-) function type.
E0520110	[Message]	Expected either a definition or a tag name.
E0520112	[Message]	Expected "while".
E0520114	[Message]	*Type* "*symbol*" was referenced but not defined.
E0520115	[Message]	A continue statement may only be used within a loop.
E0520116	[Message]	A break statement may only be used within a loop or switch.
E0520118	[Message]	A void function may not return a value.

E0520119	[Message]	Cast to type "*type*" is not allowed.
E0520120	[Message]	Return value type does not match the function type.
E0520121	[Message]	A case label may only be used within a switch.
E0520122	[Message]	A default label may only be used within a switch.
E0520123	[Message]	case label value has already appeared in this switch.
E0520124	[Message]	default label has already appeared in this switch.
E0520125	[Message]	Expected a "(".
E0520126	[Message]	Expression must be an lvalue.
E0520127	[Message]	Expected a statement.
E0520129	[Message]	A block-scope function may only have extern storage class.
E0520130	[Message]	Expected a "{".
E0520131	[Message]	Expression must have pointer-to-class type.
E0520132	[Message]	Expression must have pointer-to-struct-or-union type.
E0520133	[Message]	Expected a member name.
E0520134	[Message]	Expected a field name.
E0520135	[Message]	*symbol* has no member *member*.
E0520136	[Message]	*Type* "*symbol*" has no field "*field*".
E0520137	[Message]	Expression must be a modifiable lvalue.
E0520138	[Message]	Taking the address of a register variable is not allowed.
E0520139	[Message]	Taking the address of a bit field is not allowed.
E0520140	[Message]	Too many arguments in function call.
E0520141	[Message]	Unnamed prototyped parameters not allowed when body is present.
E0520142	[Message]	Expression must have pointer-to-object type.
E0520144	[Message]	A value of type "*type1*" cannot be used to initialize an entity of type "*type2*".
E0520145	[Message]	*Type* "*symbol*" may not be initialized.
E0520146	[Message]	Too many initializer values.
E0520147	[Message]	Declaration is incompatible with "*declaration*" (declared at line *number*).
E0520148	[Message]	*Tyep* "*symbol*" has already been initialized.
E0520149	[Message]	A global-scope declaration may not have this storage class.
E0520150	[Message]	A type name may not be redeclared as a parameter.
E0520151	[Message]	A *typedef name* may not be redeclared as a parameter.
E0520153	[Message]	Expression must have class type.
E0520154	[Message]	Expression must have struct or union type.
E0520157	[Message]	Expression must be an integral constant expression.
E0520158	[Message]	Expression must be an lvalue or a function designator.
E0520159	[Message]	Declaration is incompatible with previous "*declaration*" (declared at line *number*).

E0520160	[Message]	External name conflicts with external name of "*symbol*".
E0520165	[Message]	Too few arguments in function call.
E0520166	[Message]	Invalid floating constant.
E0520167	[Message]	Argument of type "*type1*" is incompatible with parameter of type "*type2*".
E0520168	[Message]	A function type is not allowed here.
E0520169	[Message]	Expected a declaration.
E0520170	[Message]	Pointer points outside of underlying object.
E0520171	[Message]	Invalid type conversion.
E0520172	[Message]	External/internal linkage conflict with previous declaration.
E0520173	[Message]	Floating-point value does not fit in required integral type.
E0520179	[Message]	Right operand of "%" is zero.
E0520183	[Message]	Type of cast must be integral.
E0520184	[Message]	Type of cast must be arithmetic or pointer.
E0520194	[Message]	Expected an asm string.
	[Explanation]	There is no assembler string in an "__asm()" statement.
E0520195	[Message]	An asm function must be prototyped.
E0520196	[Message]	An asm function may not have an ellipsis
E0520220	[Message]	Integral value does not fit in required floating-point type.
E0520221	[Message]	Floating-point value does not fit in required floating-point type.
E0520222	[Message]	Floating-point operation result is out of range.
E0520227	[Message]	Macro recursion.
E0520228	[Message]	Trailing comma is nonstandard.
	[Explanation]	A trailing comma is not standard.
E0520230	[Message]	Nonstandard type for a bit field.
E0520235	[Message]	Variable any-string was declared with a never-completed type.
E0520238	[Message]	Invalid specifier on a parameter.
E0520239	[Message]	Invalid specifier outside a class declaration.
E0520240	[Message]	Duplicate specifier in declaration.
E0520241	[Message]	A union is not allowed to have a base class.
E0520242	[Message]	Multiple access control specifiers are not allowed.
E0520243	[Message]	class or struct definition is missing.
E0520244	[Message]	Qualified name is not a member of class *type* or its base classes.
E0520245	[Message]	A nonstatic member reference must be relative to a specific object.
E0520246	[Message]	A nonstatic data member may not be defined outside its class.
E0520247	[Message]	*Type* "*symbol*" has already been defined.
E0520248	[Message]	Pointer to reference is not allowed.

E0520249	[Message]	Reference to reference is not allowed.
E0520250	[Message]	Reference to void is not allowed.
E0520251	[Message]	Array of reference is not allowed.
E0520252	[Message]	Reference "*name*" requires an initializer.
E0520253	[Message]	Expected a ",".
E0520254	[Message]	Type name is not allowed.
E0520255	[Message]	Type definition is not allowed.
E0520256	[Message]	Invalid redeclaration of type name "*type*".
	[Explanation]	Type name "*type*" was redeclared illegally.
E0520257	[Message]	const type "*symbol*" requires an initializer.
E0520258	[Message]	"this" may only be used inside a nonstatic member function
E0520259	[Message]	Constant value is not known.
E0520260	[Message]	Explicit type is missing ("int" assumed).
E0520262	[Message]	Not a class or struct name.
E0520263	[Message]	Duplicate base class name.
E0520264	[Message]	Invalid base class.
E0520265	[Message]	"*name*" is inaccessible.
E0520266	[Message]	"*name*" is ambiguous.
E0520267	[Message]	Old-style parameter list (anachronism).
E0520268	[Message]	Declaration may not appear after executable statement in block.
E0520269	[Message]	Conversion to inaccessible base class "*type*" is not allowed.
E0520274	[Message]	Improperly terminated macro invocation.
E0520276	[Message]	Name followed by "::" must be a class or namespace name.
E0520277	[Message]	Invalid friend declaration.
E0520278	[Message]	A constructor or destructor may not return a value.
E0520279	[Message]	Invalid destructor declaration.
E0520280	[Message]	Declaration of a member with the same name as its class.
E0520281	[Message]	Global-scope qualifier (leading "::") is not allowed.
E0520282	[Message]	The global scope has no *xxx*.
E0520283	[Message]	Qualified name is not allowed.
E0520284	[Message]	NULL reference is not allowed.
E0520285	[Message]	Initialization with "{...}" is not allowed for object of type "*type*".
E0520286	[Message]	Base class "*type*" is ambiguous.
E0520287	[Message]	Derived class *type1* contains more than one instance of class *type2*.
E0520288	[Message]	Cannot convert pointer to base class *type2* to pointer to derived class *type1* -- base class is virtual.
E0520289	[Message]	No instance of constructor *name* matches the argument list.

E0520290	[Message]	Copy constructor for class *type* is ambiguous.
E0520291	[Message]	No default constructor exists for class *type*.
E0520292	[Message]	*name* is not a nonstatic data member or base class of class *type*.
E0520293	[Message]	Indirect nonvirtual base class is not allowed.
E0520294	[Message]	Invalid union member -- class *type* has a disallowed member function.
E0520296	[Message]	Invalid use of non-lvalue array.
E0520297	[Message]	Expected an operator.
E0520298	[Message]	Inherited member is not allowed.
E0520299	[Message]	Cannot determine which instance of *name* is intended.
E0520300	[Message]	A pointer to a bound function may only be used to call the function.
E0520301	[Message]	typedef name has already been declared (with same type).
E0520302	[Message]	Symbol has already been defined.
E0520304	[Message]	No instance of *name* matches the argument list.
E0520305	[Message]	Type definition is not allowed in function return type declaration.
E0520306	[Message]	Default argument not at end of parameter list.
E0520307	[Message]	Redefinition of default argument.
E0520308	[Message]	More than one instance of *name* matches the argument list:
E0520309	[Message]	More than one instance of constructor *name* matches the argument list:
E0520310	[Message]	Default argument of type *type1* is incompatible with parameter of type *type2*.
E0520311	[Message]	Cannot overload functions distinguished by return type alone.
E0520312	[Message]	No suitable user-defined conversion from *type1* to *type2* exists.
E0520313	[Message]	Type qualifier is not allowed on this function.
E0520314	[Message]	Only nonstatic member functions may be virtual.
E0520315	[Message]	The object has cv-qualifiers that are not compatible with the member function.
E0520316	[Message]	Program too large to compile (too many virtual functions).
E0520317	[Message]	Return type is not identical to nor covariant with return type *type* of overridden virtual function *name*.
E0520318	[Message]	Override of virtual *name* is ambiguous.
E0520319	[Message]	Pure specifier ("= 0") allowed only on virtual functions.
E0520320	[Message]	Badly-formed pure specifier (only "= 0" is allowed).
E0520321	[Message]	Data member initializer is not allowed.
E0520322	[Message]	Object of abstract class type *type* is not allowed:
E0520323	[Message]	function returning abstract class *type* is not allowed:
E0520325	[Message]	inline specifier allowed on function declarations only.
E0520326	[Message]	inline is not allowed.
E0520327	[Message]	Invalid storage class for an inline function.
E0520328	[Message]	Invalid storage class for a class member.

E0520329	[Message]	Local class member *name* requires a definition.
E0520330	[Message]	*name* is inaccessible.
E0520332	[Message]	class *type* has no copy constructor to copy a const object.
E0520333	[Message]	Defining an implicitly declared member function is not allowed.
E0520334	[Message]	class *type* has no suitable copy constructor.
E0520335	[Message]	Linkage specification is not allowed.
E0520336	[Message]	Unknown external linkage specification.
E0520337	[Message]	Linkage specification is incompatible with previous "*symbol*".
E0520338	[Message]	More than one instance of overloaded function *name* has "C" linkage.
E0520339	[Message]	class *type* has more than one default constructor.
E0520340	[Message]	Value copied to temporary, reference to temporary used.
E0520341	[Message]	"operator *operator*" must be a member function.
E0520342	[Message]	Operator may not be a static member function.
E0520343	[Message]	No arguments allowed on user-defined conversion.
E0520344	[Message]	Too many parameters for this operator function.
E0520345	[Message]	Too few parameters for this operator function.
E0520346	[Message]	Nonmember operator requires a parameter with class type.
E0520347	[Message]	Default argument is not allowed.
E0520348	[Message]	More than one user-defined conversion from *type1* to *type2* applies:
E0520349	[Message]	No operator *operator* matches these operands.
E0520350	[Message]	More than one operator *operator* matches these operands:
E0520351	[Message]	First parameter of allocation function must be of type "size_t".
E0520352	[Message]	Allocation function requires "void *" return type.
E0520353	[Message]	Deallocation function requires "void" return type.
E0520354	[Message]	First parameter of deallocation function must be of type "void *".
E0520356	[Message]	Type must be an object type.
E0520357	[Message]	Base class *xxx* has already been initialized.
E0520358	[Message]	Base class name required -- *xxx* assumed (anachronism).
E0520359	[Message]	Symbol has already been initialized.
E0520360	[Message]	Name of member or base class is missing.
E0520363	[Message]	Invalid anonymous union -- nonpublic member is not allowed.
E0520364	[Message]	Invalid anonymous union -- member function is not allowed.
E0520365	[Message]	Anonymous union at global or namespace scope must be declared static.
E0520366	[Message]	Symbol provides no initializer for:
E0520367	[Message]	Implicitly generated constructor for class *type* cannot initialize:
E0520369	[Message]	*name* has an uninitialized const or reference member.

E0520371	[Message]	class *type* has no assignment operator to copy a const object.
E0520372	[Message]	class *type* has no suitable assignment operator.
E0520373	[Message]	Ambiguous assignment operator for class *type*.
E0520375	[Message]	Declaration requires a typedef name.
E0520378	[Message]	static is not allowed.
E0520380	[Message]	Expression must have pointer-to-member type.
E0520384	[Message]	No instance of overloaded *name* matches the argument list.
E0520386	[Message]	No instance of *name* matches the required type.
E0520389	[Message]	A cast to abstract class *type* is not allowed:
E0520390	[Message]	Function "main" may not be called or have its address taken.
E0520391	[Message]	A new-initializer may not be specified for an array.
E0520392	[Message]	Member function *name* may not be redeclared outside its class.
E0520393	[Message]	Pointer to incomplete class type is not allowed.
E0520394	[Message]	Reference to local variable of enclosing function is not allowed.
E0520397	[Message]	Implicitly generated assignment operator cannot copy:
E0520401	[Message]	Destructor for base class *type* is not virtual.
E0520403	[Message]	Invalid redeclaration of member "symbol".
E0520404	[Message]	Function "main" may not be declared inline.
E0520405	[Message]	Member function with the same name as its class must be a constructor.
E0520407	[Message]	A destructor may not have parameters.
E0520408	[Message]	Copy constructor for class *type1* may not have a parameter of type *type2*.
E0520409	[Message]	*Type* "*symbol*" returns incomplete type "*type*".
E0520410	[Message]	Protected *name* is not accessible through a *type* pointer or object.
E0520411	[Message]	A parameter is not allowed.
E0520413	[Message]	No suitable conversion function from *type1* to *type2* exists.
E0520415	[Message]	No suitable constructor exists to convert from *type1* to *type2*.
E0520416	[Message]	More than one constructor applies to convert from *type1* to *type2*:
E0520417	[Message]	More than one conversion function from *type1* to *type2* applies:
E0520418	[Message]	More than one conversion function from *type* to a built-in type applies:
E0520424	[Message]	A constructor or destructor may not have its address taken.
E0520427	[Message]	Qualified name is not allowed in member declaration.
E0520429	[Message]	The size of an array in "new" must be non-negative.
E0520432	[Message]	enum declaration is not allowed.
E0520433	[Message]	Qualifiers dropped in binding reference of type *type1* to initializer of type *type2*.
E0520434	[Message]	A reference of type *type1* (not const-qualified) cannot be initialized with a value of type *type2*.
E0520435	[Message]	A pointer to function may not be deleted.

E0520436	[Message]	Conversion function must be a nonstatic member function.
E0520437	[Message]	Template declaration is not allowed here.
E0520438	[Message]	Expected a "<".
E0520439	[Message]	Expected a ">".
E0520440	[Message]	Template parameter declaration is missing.
E0520441	[Message]	Argument list for "*name*" is missing.
E0520442	[Message]	Too few arguments for "*name*".
E0520443	[Message]	Too many arguments for "symbol".
E0520445	[Message]	*name1* is not used in declaring the parameter types of "*name2*".
E0520449	[Message]	More than one instance of *name* matches the required type.
E0520450	[Message]	The type "long long" is nonstandard.
E0520451	[Message]	Omission of "class" is nonstandard.
E0520452	[Message]	Return type may not be specified on a conversion function.
E0520456	[Message]	Excessive recursion at instantiation of *name*.
E0520457	[Message]	*name* is not a function or static data member.
E0520458	[Message]	Argument of type *type1* is incompatible with template parameter of type *type2*.
E0520459	[Message]	Initialization requiring a temporary or conversion is not allowed.
E0520460	[Message]	declaration of *xxx* hides function parameter.
E0520461	[Message]	Initial value of reference to non-const must be an lvalue.
E0520463	[Message]	"template" is not allowed.
E0520464	[Message]	*type* is not a class template.
E0520466	[Message]	"main" is not a valid name for a function template.
E0520467	[Message]	Invalid reference to *name* (union/nonunion mismatch).
E0520468	[Message]	A template argument may not reference a local type.
E0520469	[Message]	Tag kind of *xxx* is incompatible with declaration of "symbol".
E0520470	[Message]	The global scope has no tag named *xxx*.
E0520471	[Message]	symbol has no tag member named *xxx*.
E0520473	[Message]	*name* may be used only in pointer-to-member declaration.
E0520475	[Message]	A template argument may not reference a non-external entity.
E0520476	[Message]	Name followed by "::~" must be a class name or a type name.
E0520477	[Message]	Destructor name does not match name of class *type*.
E0520478	[Message]	Type used as destructor name does not match type *type*.
E0520481	[Message]	Invalid storage class for a template declaration.
E0520484	[Message]	Invalid explicit instantiation declaration.
E0520485	[Message]	*name* is not an entity that can be instantiated.
E0520486	[Message]	Compiler generated *name* cannot be explicitly instantiated.

E0520487	[Message]	Inline *name* cannot be explicitly instantiated.
E0520489	[Message]	*name* cannot be instantiated -- no template definition was supplied.
E0520490	[Message]	*name* cannot be instantiated -- it has been explicitly specialized.
E0520493	[Message]	No instance of *name* matches the specified type.
E0520494	[Message]	Declaring a void parameter list with a typedef is nonstandard.
E0520496	[Message]	Template parameter *name* may not be redeclared in this scope.
E0520498	[Message]	Template argument list must match the parameter list.
E0520500	[Message]	Extra parameter of postfix "operator *xxx*" must be of type "int".
E0520501	[Message]	An operator name must be declared as a function.
E0520502	[Message]	Operator name is not allowed.
E0520503	[Message]	*name* cannot be specialized in the current scope.
E0520504	[Message]	Nonstandard form for taking the address of a member function.
E0520505	[Message]	Too few template parameters -- does not match previous declaration.
E0520506	[Message]	Too many template parameters -- does not match previous declaration.
E0520507	[Message]	Function template for operator delete(void *) is not allowed.
E0520508	[Message]	class template and template parameter may not have the same name.
E0520510	[Message]	A template argument may not reference an unnamed type.
E0520511	[Message]	Enumerated type is not allowed.
E0520513	[Message]	A value of type "type1" cannot be assigned to an entity of type "type2".
E0520515	[Message]	Cannot convert to incomplete class *type*.
E0520516	[Message]	const object requires an initializer.
E0520517	[Message]	Object has an uninitialized const or reference member.
E0520518	[Message]	Nonstandard preprocessing directive.
E0520519	[Message]	*name* may not have a template argument list.
E0520520	[Message]	Initialization with "{...}" expected for aggregate object.
E0520521	[Message]	Pointer-to-member selection class types are incompatible (*type1* and *type2*).
E0520525	[Message]	A dependent statement may not be a declaration.
	[Explanation]	Cannot write declaration due to lack of "{" character after "if()" statement.
E0520526	[Message]	A parameter may not have void type.
E0520529	[Message]	This operator is not allowed in a template argument expression.
E0520530	[Message]	Try block requires at least one handler/
E0520531	[Message]	Handler requires an exception declaration.
E0520532	[Message]	Handler is masked by default handler.
E0520536	[Message]	Exception specification is incompatible with that of previous *name*.
E0520540	[Message]	Support for exception handling is disabled.
E0520543	[Message]	Non-arithmetic operation not allowed in nontype template argument.

E0520544	[Message]	Use of a local type to declare a nonlocal variable.
E0520545	[Message]	Use of a local type to declare a function.
E0520546	[Message]	Transfer of control bypasses initialization of:
E0520548	[Message]	Transfer of control into an exception handler.
E0520551	[Message]	symbol cannot be defined in the current scope.
E0520555	[Message]	Tag kind of *name* is incompatible with template parameter of type *type*.
E0520556	[Message]	Function template for operator new(size_t) is not allowed.
E0520558	[Message]	Pointer to member of type "*type*" is not allowed.
E0520559	[Message]	Tointer to member of type *type* is not allowed.
E0520560	[Message]	symbol is reserved for future use as a keyword.
E0520561	[Message]	Invalid macro definition:
E0520562	[Message]	Invalid macro undefinition:
E0520598	[Message]	A template parameter may not have void type.
E0520599	[Message]	Excessive recursive instantiation of *name* due to instantiate-all mode.
E0520601	[Message]	A throw expression may not have void type.
E0520603	[Message]	Parameter of abstract class type *type* is not allowed:
E0520604	[Message]	Array of abstract class *type* is not allowed:
E0520605	[Message]	Floating-point template parameter is nonstandard.
E0520606	[Message]	This pragma must immediately precede a declaration.
E0520607	[Message]	This pragma must immediately precede a statement.
E0520608	[Message]	This pragma must immediately precede a declaration or statement.
E0520609	[Message]	This kind of pragma may not be used here.
E0520612	[Message]	Specific definition of inline template function must precede its first use.
E0520615	[Message]	Parameter type involves pointer to array of unknown bound.
E0520616	[Message]	Parameter type involves reference to array of unknown bound.
E0520618	[Message]	struct or union declares no named members.
E0520619	[Message]	Nonstandard unnamed field.
E0520620	[Message]	Nonstandard unnamed member.
E0520643	[Message]	restrict is not allowed.
E0520644	[Message]	A pointer or reference to function type may not be qualified by "restrict".
E0520647	[Message]	Conflicting calling convention modifiers.
E0520651	[Message]	A calling convention may not be followed by a nested declarator.
E0520654	[Message]	Declaration modifiers are incompatible with previous declaration.
E0520656	[Message]	Transfer of control into a try block.
E0520658	[Message]	Closing brace of template definition not found.
E0520660	[Message]	Invalid packing alignment value.

E0520661	[Message]	Expected an integer constant.
E0520663	[Message]	Invalid source file identifier string.
E0520664	[Message]	A class template cannot be defined in a friend declaration.
E0520665	[Message]	asm is not allowed.
E0520666	[Message]	asm must be used with a function definition.
E0520667	[Message]	asm function is nonstandard.
E0520668	[Message]	Ellipsis with no explicit parameters is nonstandard.
E0520669	[Message]	&... is nonstandard.
E0520670	[Message]	invalid use of "&...".
E0520673	[Message]	A reference of type *type1* cannot be initialized with a value of type *type2*.
E0520674	[Message]	Initial value of reference to const volatile must be an lvalue.
E0520676	[Message]	Using out-of-scope declaration of *type* "*symbol*" (declared at line *number*).
E0520691	[Message]	*xxx*, required for copy that was eliminated, is inaccessible.
E0520692	[Message]	*xxx* required for copy that was eliminated, is not callable because reference parameter cannot be bound to rvalue.
E0520693	[Message]	<typeinfo> must be included before typeid is used.
E0520694	[Message]	*xxx* cannot cast away const or other type qualifiers.
E0520695	[Message]	The type in a dynamic_cast must be a pointer or reference to a complete class type, or void *.
E0520696	[Message]	The operand of a pointer dynamic_cast must be a pointer to a complete class type.
E0520697	[Message]	The operand of a reference dynamic_cast must be an lvalue of a complete class type.
E0520698	[Message]	The operand of a runtime dynamic_cast must have a polymorphic class type.
E0520701	[Message]	An array type is not allowed here.
E0520702	[Message]	Expected an "=".
E0520703	[Message]	Expected a declarator in condition declaration.
E0520704	[Message]	*xxx*, declared in condition, may not be redeclared in this scope.
E0520705	[Message]	Default template arguments are not allowed for function templates.
E0520706	[Message]	Expected a "," or ">".
E0520707	[Message]	Expected a template parameter list.
E0520709	[Message]	bool type is not allowed.
E0520710	[Message]	Offset of base class *name1* within class *name2* is too large.
E0520711	[Message]	Expression must have bool type (or be convertible to bool).
E0520717	[Message]	The type in a const_cast must be a pointer, reference, or pointer to member to an object type.
E0520718	[Message]	A const_cast can only adjust type qualifiers; it cannot change the underlying type.
E0520719	[Message]	mutable is not allowed.
E0520724	[Message]	namespace definition is not allowed.

E0520725	[Message]	name must be a namespace name.
E0520726	[Message]	namespace alias definition is not allowed.
E0520727	[Message]	namespace-qualified name is required.
E0520728	[Message]	A namespace name is not allowed.
E0520730	[Message]	*name* is not a class template.
E0520731	[Message]	Array with incomplete element type is nonstandard.
E0520732	[Message]	Allocation operator may not be declared in a namespace.
E0520733	[Message]	Deallocation operator may not be declared in a namespace.
E0520734	[Message]	*name1* conflicts with using-declaration of *name2*.
E0520735	[Message]	using-declaration of *name1* conflicts with *name2*.
E0520742	[Message]	symbol has no actual member *xxx*.
E0520749	[Message]	A type qualifier is not allowed.
E0520750	[Message]	*name* was used before its template was declared.
E0520751	[Message]	Static and nonstatic member functions with same parameter types cannot be overloaded.
E0520752	[Message]	No prior declaration of "symbol".
E0520753	[Message]	A template-id is not allowed.
	[Explanation]	The use of templates (*template name*<template *argument*>) is not allowed.
E0520754	[Message]	A class-qualified name is not allowed.
E0520755	[Message]	symbol may not be redeclared in the current scope.
E0520756	[Message]	Qualified name is not allowed in namespace member declaration.
E0520757	[Message]	symbol is not a type name.
E0520758	[Message]	Explicit instantiation is not allowed in the current scope.
E0520759	[Message]	*symbol* cannot be explicitly instantiated in the current scope.
E0520761	[Message]	typename may only be used within a template.
E0520765	[Message]	Nonstandard character at start of object-like macro definition.
E0520766	[Message]	Exception specification for virtual *name1* is incompatible with that of overridden *name2*.
E0520767	[Message]	Conversion from pointer to smaller integer.
E0520768	[Message]	Exception specification for implicitly declared virtual *name1* is incompatible with that of overridden *name2*.
E0520769	[Message]	*name1*, implicitly called from *name2*, is ambiguous.
E0520771	[Message]	"explicit" is not allowed.
E0520772	[Message]	Declaration conflicts with *xxx* (reserved class name).
E0520773	[Message]	Only "()" is allowed as initializer for array "symbol".
E0520774	[Message]	"virtual" is not allowed in a function template declaration.
E0520775	[Message]	Invalid anonymous union -- class member template is not allowed.

E0520776	[Message]	Template nesting depth does not match the previous declaration of %n.
E0520779	[Message]	*xxx*, declared in for-loop initialization, may not be redeclared in this scope.
E0520782	[Message]	Definition of virtual *name* is required here.
E0520784	[Message]	A storage class is not allowed in a friend declaration.
E0520785	[Message]	Template parameter list for *name* is not allowed in this declaration.
E0520786	[Message]	*name* is not a valid member class or function template.
E0520787	[Message]	Not a valid member class or function template declaration.
E0520788	[Message]	A template declaration containing a template parameter list may not be followed by an explicit specialization declaration.
E0520789	[Message]	Explicit specialization of *name1* must precede the first use of *name2*.
E0520790	[Message]	Explicit specialization is not allowed in the current scope.
E0520791	[Message]	Partial specialization of *name* is not allowed.
E0520792	[Message]	*name* is not an entity that can be explicitly specialized.
E0520793	[Message]	Explicit specialization of %n must precede its first use.
E0520795	[Message]	Specializing *name* requires "template<>" syntax.
E0520799	[Message]	Specializing *symbol* without "template<>" syntax is nonstandard.
E0520800	[Message]	This declaration may not have extern "C" linkage.
E0520801	[Message]	*name* is not a class or function template name in the current scope.
E0520803	[Message]	Specifying a default argument when redeclaring an already referenced function template is not allowed.
E0520804	[Message]	Cannot convert pointer to member of base class type*2* to pointer to member of derived class type*1* -- base class is virtual.
E0520805	[Message]	Exception specification is incompatible with that of *name.*
E0520807	[Message]	Unexpected end of default argument expression.
E0520808	[Message]	Default-initialization of reference is not allowed.
E0520809	[Message]	Uninitialized "symbol" has a const member.
E0520810	[Message]	Uninitialized base class type has a const member.
E0520811	[Message]	const *name* requires an initializer -- class *type* has no explicitly declared default constructor.
E0520812	[Message]	Const object requires an initializer -- class *type* has no explicitly declared default constructor.
E0520816	[Message]	In a function definition a type qualifier on a "void" return type is not allowed.
E0520817	[Message]	Static data member declaration is not allowed in this class.
E0520818	[Message]	Template instantiation resulted in an invalid function declaration.
E0520819	[Message]	... is not allowed.
E0520822	[Message]	Invalid destructor name for type *type*.
E0520824	[Message]	Destructor reference is ambiguous -- both *name1* and *name2* could be used.
E0520827	[Message]	Only one member of a union may be specified in a constructor initializer list.

E0520828	[Message]	Support for "new[]" and "delete[]" is disabled.
E0520832	[Message]	No appropriate operator delete is visible.
E0520833	[Message]	Pointer or reference to incomplete type is not allowed.
E0520834	[Message]	Invalid partial specialization -- *name* is already fully specialized.
E0520835	[Message]	Incompatible exception specifications.
E0520840	[Message]	A template argument list is not allowed in a declaration of a primary template.
E0520841	[Message]	Partial specializations may not have default template arguments.
E0520842	[Message]	*name1* is not used in or cannot be deduced from the template argument list of *name2*.
E0520844	[Message]	The template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter.
E0520845	[Message]	This partial specialization would have been used to instantiate *name*.
E0520846	[Message]	This partial specialization would have made the instantiation of *name* ambiguous.
E0520847	[Message]	Expression must have integral or enum type.
E0520848	[Message]	Expression must have arithmetic or enum type.
E0520849	[Message]	Expression must have arithmetic, enum, or pointer type.
E0520850	[Message]	Type of cast must be integral or enum.
E0520851	[Message]	Type of cast must be arithmetic, enum, or pointer.
E0520852	[Message]	Expression must be a pointer to a complete object type.
E0520854	[Message]	A partial specialization nontype argument must be the name of a nontype parameter or a constant.
E0520855	[Message]	Return type is not identical to return type *type* of overridden virtual function *name*.
E0520857	[Message]	A partial specialization of a class template must be declared in the namespace of which it is a member.
E0520858	[Message]	*name* is a pure virtual function.
E0520859	[Message]	Pure virtual *name* has no overrider.
E0520861	[Message]	Invalid character in input line.
E0520862	[Message]	Function returns incomplete type "*type*".
E0520864	[Message]	*name* is not a template.
E0520865	[Message]	A friend declaration may not declare a partial specialization.
E0520868	[Message]	Space required between adjacent ">" delimiters of nested template argument lists (">>" is the right shift operator).
E0520870	[Message]	Invalid multibyte character sequence.
E0520871	[Message]	Template instantiation resulted in unexpected function type of *type1* (the meaning of a name may have changed since the template declaration -- the type of the template is *type2*).
E0520872	[Message]	Ambiguous guiding declaration -- more than one function template *name* matches type *type*.
E0520873	[Message]	Non-integral operation not allowed in nontype template argument.
E0520875	[Message]	Embedded C++ does not support templates.

E0520876	[Message]	Embedded C++ does not support exception handling.
E0520877	[Message]	Embedded C++ does not support namespaces.
E0520878	[Message]	Embedded C++ does not support run-time type information.
E0520879	[Message]	Embedded C++ does not support the new cast syntax.
E0520880	[Message]	Embedded C++ does not support using-declarations.
E0520881	[Message]	Embedded C++ does not support \"mutable\".
E0520882	[Message]	Embedded C++ does not support multiple or virtual inheritance.
E0520885	[Message]	*type1* cannot be used to designate constructor for *type2*.
E0520886	[Message]	Invalid suffix on integral constant.
	[Explanation]	The integer constant has an invalid suffix.
E0520890	[Message]	Variable length array with unspecified bound is not allowed.
E0520891	[Message]	An explicit template argument list is not allowed on this declaration.
E0520892	[Message]	An entity with linkage cannot have a type involving a variable length array.
E0520893	[Message]	A variable length array cannot have static storage duration.
E0520894	[Message]	Entity-kind "*name*" is not a template.
E0520896	[Message]	Expected a template argument.
E0520898	[Message]	Nonmember operator requires a parameter with class or enum type.
E0520901	[Message]	Qualifier of destructor name *type1* does not match type *type2*.
E0520915	[Message]	A segment name has already been specified.
E0520916	[Message]	Cannot convert pointer to member of derived class *type1* to pointer to member of base class *type2* -- base class is virtual.
E0520928	[Message]	Incorrect use of va_start.
E0520929	[Message]	Incorrect use of va_arg.
E0520930	[Message]	Incorrect use of va_end.
E0520934	[Message]	A member with reference type is not allowed in a union.
E0520935	[Message]	Typedef may not be specified here.
E0520937	[Message]	A class or namespace qualified name is required.
E0520938	[Message]	Return type "int" omitted in declaration of function "main".
E0520939	[Message]	Pointer-to-member representation *xxx* is too restrictive for *xxx*.
E0520940	[Message]	Missing return statement at end of non-void *type* "*symbol*".
E0520946	[Message]	Name following "template" must be a template.
E0520948	[Message]	Nonstandard local-class friend declaration -- no prior declaration in the enclosing scope.
E0520951	[Message]	Return type of function "main" must be "int".
E0520952	[Message]	A nontype template parameter may not have class type.
E0520953	[Message]	A default template argument cannot be specified on the declaration of a member of a class template outside of its class.

E0520954	[Message]	A return statement is not allowed in a handler of a function try block of a constructor.
E0520955	[Message]	Ordinary and extended designators cannot be combined in an initializer designation.
E0520956	[Message]	The second subscript must not be smaller than the first.
E0520960	[Message]	Type used as constructor name does not match type *type*.
E0520961	[Message]	Use of a type with no linkage to declare a variable with linkage.
E0520962	[Message]	Use of a type with no linkage to declare a function.
E0520963	[Message]	Return type may not be specified on a constructor.
E0520964	[Message]	Return type may not be specified on a destructor.
E0520965	[Message]	Incorrectly formed universal character name.
E0520966	[Message]	Universal character name specifies an invalid character.
E0520967	[Message]	A universal character name cannot designate a character in the basic character set.
E0520968	[Message]	This universal character is not allowed in an identifier.
E0520969	[Message]	The identifier __VA_ARGS__ can only appear in the replacement lists of variadic macros.
E0520971	[Message]	Array range designators cannot be applied to dynamic initializers.
E0520972	[Message]	Property name cannot appear here.
E0520975	[Message]	A variable-length array type is not allowed.
E0520976	[Message]	A compound literal is not allowed in an integral constant expression.
E0520977	[Message]	A compound literal of type "type" is not allowed.
E0520978	[Message]	A template friend declaration cannot be declared in a local class.
E0520979	[Message]	Ambiguous "?" operation: second operand of type *type1* can be converted to third operand type *type2*, and vice versa.
E0520980	[Message]	Call of an object of a class type without appropriate operator() or conversion functions to pointer-to-function type.
E0520982	[Message]	There is more than one way an object of type "type" can be called for the argument list:
E0520983	[Message]	typedef name has already been declared (with similar type).
E0520985	[Message]	Storage class "mutable" is not allowed for anonymous unions.
E0520987	[Message]	Abstract class type *type* is not allowed as catch type:
E0520988	[Message]	A qualified function type cannot be used to declare a nonmember function or a static member function.
E0520989	[Message]	A qualified function type cannot be used to declare a parameter.
E0520990	[Message]	Cannot create a pointer or reference to qualified function type.
E0520992	[Message]	Invalid macro definition:.
E0520993	[Message]	Subtraction of pointer types "*type1*" and "*type2*" is nonstandard.
E0520994	[Message]	An empty template parameter list is not allowed in a template template parameter declaration.

E0520995	[Message]	Expected "class".
E0520996	[Message]	The "class" keyword must be used when declaring a template template parameter.
E0520998	[Message]	A qualified name is not allowed for a friend declaration that is a function definition.
E0520999	[Message]	*symbol1* is not compatible with "*symbol2*".
E0521001	[Message]	Class member designated by a using-declaration must be visible in a direct base class.
E0521006	[Message]	A template template parameter cannot have the same name as one of its template parameters.
E0521007	[Message]	Recursive instantiation of default argument.
E0521009	[Message]	*symbol* is not an entity that can be defined.
E0521010	[Message]	Destructor name must be qualified.
E0521011	[Message]	Friend class name may not be introduced with "typename".
E0521012	[Message]	A using-declaration may not name a constructor or destructor.
E0521013	[Message]	A qualified friend template declaration must refer to a specific previously declared template.
E0521014	[Message]	Invalid specifier in class template declaration.
E0521015	[Message]	Argument is incompatible with formal parameter.
E0521017	[Message]	Loop in sequence of "operator->" functions starting at class *xxx*.
E0521018	[Message]	*xxx* has no member class *xxx*.
E0521019	[Message]	The global scope has no class named *xxx*.
E0521020	[Message]	Recursive instantiation of template default argument.
E0521021	[Message]	Access declarations and using-declarations cannot appear in unions.
E0521022	[Message]	*xxx* is not a class member.
E0521023	[Message]	Nonstandard member constant declaration is not allowed.
E0521029	[Message]	Type containing an unknown-size array is not allowed.
E0521030	[Message]	A variable with static storage duration cannot be defined within an inline function.
E0521031	[Message]	An entity with internal linkage cannot be referenced within an inline function with external linkage.
E0521032	[Message]	Argument type %t does not match this type-generic function macro.
E0521034	[Message]	Friend declaration cannot add default arguments to previous declaration.
E0521035	[Message]	*xxx* cannot be declared in this scope.
E0521036	[Message]	The reserved identifier "*symbol*" may only be used inside a function.
E0521037	[Message]	This universal character cannot begin an identifierl.
E0521038	[Message]	Expected a string literal.
E0521039	[Message]	Unrecognized STDC pragma.
E0521040	[Message]	Expected "ON", "OFF", or "DEFAULT".
E0521041	[Message]	A STDC pragma may only appear between declarations in the global scope or before any statements or declarations in a block scope.

E0521042	[Message]	Incorrect use of va_copy.
E0521043	[Message]	*xxx* can only be used with floating-point types.
E0521044	[Message]	Complex type is not allowed.
E0521045	[Message]	Invalid designator kind.
E0521047	[Message]	Complex floating-point operation result is out of range.
E0521048	[Message]	Conversion between real and imaginary yields zero.
E0521049	[Message]	An initializer cannot be specified for a flexible array member.
E0521051	[Message]	Standard requires that "symbol" be given a type by a subsequent declaration ("int" assumed).
E0521052	[Message]	A definition is required for inline "symbol".
E0521054	[Message]	A floating-point type must be included in the type specifier for a _Complex or _Imaginary type.
E0521055	[Message]	Types cannot be declared in anonymous unions.
E0521056	[Message]	Returning pointer to local variable.
E0521057	[Message]	Returning pointer to local temporary.
E0521061	[Message]	Declaration of "symbol" is incompatible with a declaration in another translation unit.
E0521062	[Message]	The other declaration is %p.
E0521065	[Message]	A field declaration cannot have a type involving a variable length array.
E0521066	[Message]	Declaration of "*symbol*" had a different meaning during compilation of *file*.
E0521067	[Message]	Eexpected "template".
E0521072	[Message]	A declaration cannot have a label.
E0521075	[Message]	"*symbol*" already defined during compilation of any-string.s
E0521076	[Message]	"*symbol*" already defined in another translation unit.
E0521081	[Message]	A field with the same name as its class cannot be declared in a class with a user-declared constructor.
E0521086	[Message]	The object has cv-qualifiers that are not compatible with the member "*symbol*".
E0521087	[Message]	No instance of *xxx* matches the argument list and object (the object has cv-qualifiers that prevent a match).
E0521088	[Message]	An attribute specifies a mode incompatible with *xxx*.
E0521089	[Message]	There is no type with the width specified.
E0521139	[Message]	The "template" keyword used for syntactic disambiguation may only be used within a template.
E0521144	[Message]	Storage class must be auto or register.
E0521146	[Message]	*xxx* is not a base class member.
E0521158	[Message]	void return type cannot be qualified.
E0521161	[Message]	A member template corresponding to *xxx* is declared as a template of a different kind in another translation unit.
E0521163	[Message]	va_start should only appear in a function with an ellipsis parameter.

E0521201	[Message]	typedef *xxx* may not be used in an elaborated type specifier.
E0521203	[Message]	Parameter *parameter* may not be redeclared in a catch clause of function try block.
E0521204	[Message]	The initial explicit specialization of *xxx* must be declared in the namespace containing the template.
E0521206	[Message]	"template" must be followed by an identifier.
E0521227	[Message]	Transfer of control into a statement expression is not allowed.
E0521229	[Message]	This statement is not allowed inside of a statement expression.
E0521230	[Message]	Anon-POD class definition is not allowed inside of a statement expression.
E0521254	[Message]	Integer overflow in internal computation due to size or complexity of "*type*".
E0521255	[Message]	Integer overflow in internal computation.
E0521273	[Message]	Alignment-of operator applied to incomplete type.
E0521280	[Message]	Conversion from inaccessible base class *xxx* is not allowed.
E0521282	[Message]	String literals with different character kinds cannot be concatenated.
E0521291	[Message]	A non-POD class type cannot be fetched by va_arg.
E0521292	[Message]	The 'u' or 'U' suffix must appear before the 'l' or 'L' suffix in a fixed-point literal.
E0521295	[Message]	Fixed-point constant is out of range.
E0521303	[Message]	Expression must have integral, enum, or fixed-point type.
E0521304	[Message]	Expression must have integral or fixed-point type.
E0521311	[Message]	Fixed-point types have no classification.
E0521312	[Message]	A template parameter may not have fixed-point type.
E0521313	[Message]	Hexadecimal floating-point constants are not allowed.
E0521315	[Message]	Floating-point value does not fit in required fixed-point type.
E0521317	[Message]	Fixed-point conversion resulted in a change of sign.
E0521318	[Message]	Integer value does not fit in required fixed-point type.
E0521319	[Message]	Fixed-point operation result is out of range.
E0521320	[Message]	Multiple named address spaces.
E0521321	[Message]	Variable with automatic storage duration cannot be stored in a named address space.
E0521322	[Message]	Type cannot be qualified with named address space.
E0521323	[Message]	Function type cannot be qualified with named address space.
E0521324	[Message]	Field type cannot be qualified with named address space.
E0521325	[Message]	Fixed-point value does not fit in required floating-point type.
E0521326	[Message]	Fixed-point value does not fit in required integer type.
E0521327	[Message]	Value does not fit in required fixed-point type.
E0521344	[Message]	A named address space qualifier is not allowed here.
E0521345	[Message]	An empty initializer is invalid for an array with unspecified bound.
E0521348	[Message]	Declaration hides "*symbol*".

E0521349	[Message]	A parameter cannot be allocated in a named address space.
E0521350	[Message]	Invalid suffix on fixed-point or floating-point constant.
E0521351	[Message]	A register variable cannot be allocated in a named address space.
E0521352	[Message]	Expected "SAT" or "DEFAULT".
E0521355	[Message]	A function return type cannot be qualified with a named address space.
E0521365	[Message]	Named-register variables cannot have void type.
E0521372	[Message]	Nonstandard qualified name in global scope declaration.
E0521380	[Message]	Virtual *xxx* was not defined (and cannot be defined elsewhere because it is a member of an unnamed namespace).
E0521381	[Message]	Carriage return character in source line outside of comment or character/string literal.
	[Explanation]	Carriage return character (\r) in source line outside of comment or character/string literal.
E0521382	[Message]	Expression must have fixed-point type.
E0521398	[Message]	Invalid member for anonymous member class -- class *xxx* has a disallowed member function.
E0521403	[Message]	A variable-length array is not allowed in a function return type.
E0521404	[Message]	Variable-length array type is not allowed in pointer to member of type "*type*".
E0521405	[Message]	The result of a statement expression cannot have a type involving a variable-length array.
E0521420	[Message]	Some enumerator values cannot be represented by the integral type underlying the enum type.
E0521424	[Message]	Second operand of offsetof must be a field.
E0521425	[Message]	Second operand of offsetof may not be a bit field.
E0521436	[Message]	*xxx* is only allowed in C.
E0521437	[Message]	__ptr32 and __ptr64 must follow a "*".
E0521441	[Message]	Complex integral types are not supported.
E0521442	[Message]	__real and __imag can only be applied to complex values.
E0521445	[Message]	Invalid redefinition of "*symbol*".
E0521534	[Message]	Duplicate function modifier.
E0521535	[Message]	Invalid character for char16_t literal.
E0521536	[Message]	__LPREFIX cannot be applied to char16_t or char32_t literals.
E0521537	[Message]	Unrecognized calling convention *xxx* must be one of:
E0521539	[Message]	Option "--uliterals" can be used only when compiling C.
E0521542	[Message]	Some enumerator constants cannot be represented by "type".
E0521543	[Message]	*xxx* not allowed in current mode.
E0521557	[Message]	Alias creates cycle of aliased entities.
E0521558	[Message]	Subscript must be constant.
E0521574	[Message]	Static assertion failed with *xxx*.

E0521576	[Message]	Field name resolves to more than one offset -- see "*symbol1*" and "*symbol2*".
E0521577	[Message]	*xxx* is not a field name.
E0521578	[Message]	case label value has already appeared in this switch at line *number*.
E0521582	[Message]	The option to list macro definitions may not be specified when compiling more than one translation unit.
E0521583	[Message]	Unexpected parenthesis after declaration of "symbol" (malformed parameter list or invalid initializer?).
E0521584	[Message]	Parentheses around a string initializer are nonstandard.
E0521586	[Message]	A variable declared with an auto type specifier cannot appear in its own initializer.
E0521587	[Message]	Cannot deduce "auto" type.
E0521588	[Message]	Initialization with "{...}" is not allowed for "auto" type.
E0521589	[Message]	auto type cannot appear in top-level array type.
E0521590	[Message]	auto type cannot appear in top-level function type.
E0521593	[Message]	Cannot deduce "auto" type (initializer required).
E0521596	[Message]	Invalid use of a type qualifier.
E0521597	[Message]	A union cannot be abstract or sealed.
E0521598	[Message]	auto is not allowed here.
E0521602	[Message]	struct/union variable "*variable*" with a member of incomplete type cannot be placed into the section.
E0521603	[Message]	Variable of incomplete type "*variable*" cannot be placed into the section.
E0521604	[Message]	Illegal section attribute.
E0521605	[Message]	Illegal #pragma *character string* syntax.
E0521606	[Message]	"*function*" has already been placed into another section.
	[Explanation]	A "#pragma text" has already been specified for function "*function*". It cannot be put into a different section.
E0521608	[Message]	#pragma asm is not allowed outside of function.
E0521609	[Message]	The #pragma endasm for this #pragma asm is missing.
E0521610	[Message]	The #pragma asm for this #pragma endasm is missing.
E0521612	[Message]	Duplicate interrupt hander for "*request*".
E0521613	[Message]	Interrupt request name "*request*" not supported.
E0521614	[Message]	Duplicate #pragma interrupt for this function.
E0521615	[Message]	Duplicate #pragma smart_correct for this function "*function*".
	[Explanation]	A "#pragma smart_correct" has already been specified for function "*function*".
E0521616	[Message]	*Type* "*symbol*" has already been placed into another section (declared as extern).
E0521617	[Message]	*Type* "*symbol*" has already been placed into another section.
E0521618	[Message]	*Type* "*symbol*" has already been declared with #pragma section.
E0521619	[Message]	*Type* "*symbol*" has already been declared without #pragma section.
E0521620	[Message]	"*function*()" argument overflow. use "*minimum value - maximum value*".

E0521621	[Message]	Cannot write I/O register "*register name*".
E0521622	[Message]	Cannot read I/O register "*register name*".
E0521623	[Message]	Cannot use *expanded specification*. Device must be specified.
E0521624	[Message]	Second argument for __set_il()must be string literal.
E0521625	[Message]	Cannot set interrupt level for "*request*".
E0521626	[Message]	*Specification character string* is specified for function "*function name*", previously specified #pragma inline is ignored.
E0521627	[Message]	Function for #pragma smart_correct is same.
E0521628	[Message]	Function for #pragma smart_correct "*function*" is undefined.
E0521629	[Message]	Could not open symbol file "*file name*".
E0521630	[Message]	Could not close symbol file "*file name*".
E0521631	[Message]	Syntax error in symbol file.
E0521632	[Message]	Unrecognized symbol information "*character string*" is ignored.
E0521633	[Message]	Section name is not specified.
E0521634	[Message]	Unrecognized section name "*section*".
E0521635	[Message]	"*variable name*" has already been placed into "*section name*" section in symbol file. The latter is ignored.
E0521636	[Message]	"*variable name*" has already been placed into "*section name*" section in symbol file. #pragma is ignored.
E0521637	[Message]	Illegal binary digit.
E0521638	[Message]	First argument for *special function name*()must be integer constant.
E0521639	[Message]	Function "*function name*" specified as "direct" can not be allocated in text.
E0521640	[Message]	Function allocated in text can not be specified #pragma interrupt with "direct".
E0521641	[Message]	FE level interrupt not supported.
E0521642	[Message]	Cannnot give a name for "*attribute*" section.
E0521643	[Message]	"direct" cannot be specified for plural interrupt.
E0521644	[Message]	Reduced exception handler option of device is available. Address of the handler-maybe overlaps.
E0521645	[Message]	Function "*function name*" has illegal type for interrupt function,must be void(void).
E0521646	[Message]	Cannot use direct with NO_VECT.
E0521647	[Message]	*character string* is not allowed here.
E0521648	[Message]	Cannot call *type* function "*function name*".
E0521649	[Message]	Cannot use *character string1* with *character string2*.
	[Explanation]	The functions of *string 1* and *string 2* cannot be used at the same time.
	[Message]	[CC-RX] White space is required between the macro name *xxx* and its replacement text.
	[Action by User]	[CC-RX] Insert white space between the macro name and its replacement text.

E0521650	[Message]	*type* "*symbol name*" has already been declared with other #pragma pic/nopic.
	[Explanation]	There is a "#pragma pic/nopic" specification in conflict with *type* "*symbol name*".
E0523005	[Message]	Invalid pragma declaration
	[Explanation]	Write the #pragma syntax in accord with the correct format.
E0523006	[Message]	"*symbol name*" has already been specified by other pragma
	[Explanation]	Two or more #pragma directives have been specified for one symbol, and such specification is not allowed.
E0523007	[Message]	Pragma may not be specified after definition
	[Explanation]	The #pragma directive precedes definition of the target symbol.
E0523008	[Message]	Invalid kind of pragma is specified to this symbol
	[Explanation]	The given type of #pragma directive is not specifiable for the symbol.
E0523042	[Message]	Using "*function item*" function at influence the code generation of "SuperH" compiler
	[Explanation]	The use of "*function item*" may affect compatibility with the SuperH compiler. Confirm details of differences from the specification.
E0523047	[Message]	Illegal #pragma interrupt declaration
	[Explanation]	The interrupt function declaration by #pragma interrupt is incorrect.
E0523049	[Message]	Multiple #pragma for one function
	[Explanation]	Two or more #pragma directives have been specified for one symbol, and such specification is not allowed.
E0523057	[Message]	Illegal section specified
	[Explanation]	Strings that are not usable for the purpose were used to specify the attributes of sections.
E0523058	[Message]	Illegal #pragma section syntax
	[Explanation]	The #pragma section syntax is illegal.
E0523059	[Message]	Cannot change text section
	[Explanation]	The #pragma section syntax is incorrect.
E0523061	[Message]	Argument is incompatible with formal parameter of intrinsic function.
E0523062	[Message]	Return value type does not match the intrinsic function type.
E0523065	[Message]	Cannot assign address constant to initializer for bitfield
	[Explanation]	An address constant cannot be written as the initial value of a bit field.
E0523066	[Message]	The combination of the option and section specification is inaccurate
E0523129	[Message]	The "*option name*" option is necessary for use of "*function*".
	[Explanation]	The option must be specified to use this function.
E0532002	[Message]	Exception *exception* has occurred at compile time.
E0544003	[Message]	The size of "*section name*" section exceeds the limit.
E0544240	[Message]	Illegal naming of section "*section name*".
	[Explanation]	There is an error in section naming. The same section name is specified for different use of the section.

E0544854	[Message]	Illegal address was specified with #pragma address.
	[Explanation]	"#pragma address" specification satisfies either of the following conditions. (1) The same address was specified for different variables. (2) Overlapping address ranges were specified for different variables.
E0552000	[Message]	No space after mnemonic or directive.
	[Explanation]	The mnemonic or assemble directive is not followed by a space character.
	[Action by User]	Enter a space character between the instruction and operand.
E0552001	[Message]	',' is missing.
	[Explanation]	',' is not entered.
	[Action by User]	Insert a comma to separate between operands.
E0552002	[Message]	Characters exist in expression.
	[Explanation]	Extra characters are written in an instruction or expression.
	[Action by User]	Check the rules to be followed when writing an expression.
E0552003	[Message]	Size specifier is missing.
	[Explanation]	No size specifier is entered.
	[Action by User]	Write a size specifier.
E0552004	[Message]	Invalid operand(s) exist in instruction.
	[Explanation]	The instruction contains an invalid operand.
	[Action by User]	Check the syntax for this instruction and rewrite it correctly.
E0552005	[Message]	Operand type is not appropriate.
	[Explanation]	The operand type is incorrect.
	[Action by User]	Check the syntax for this operand and rewrite it correctly.
E0552006	[Message]	Size specifier is not appropriate.
	[Explanation]	The size specifier is written incorrectly.
	[Action by User]	Rewrite the size specifier correctly.
E0552007	[Message]	Operand label is not in the same section.
	[Explanation]	The branch destination is not in the same section.
	[Action by User]	Execution can branch only to a destination within the same section. Correct the mnemonic.
E0552008	[Message]	Illegal displacement value.
	[Explanation]	An illegal displacement value is specified.
	[Action by User]	Specify a multiple of 2 when the size specifier is W. Specify a multiple of 4 when the size specifier is L.
E0552009	[Message]	FPU instruction or FPSW is used.
	[Explanation]	A floating-point operation (FPU) instruction or FPSW is used.
	[Action by User]	Check the CPU type.

E0552010	[Message]	ISAV2 instruction or EXTB is used
	[Action by User]	Check the RX instruction-set architecture selected by the -isa option or environment variable ISA_RX.
E0552011	[Message]	*Type* instruction is used.
	[Action by User]	Instructions of the indicated *type* are not usable with the current settings of options or environment variables. Change the settings as required.
E0552020	[Message]	Invalid operand(s) exist in debug information
	[Explanation]	The operand of the .line directive is invalid.
E0552022	[Message]	Symbol name is missing.
	[Explanation]	Symbol is not entered.
	[Action by User]	Write a symbol name.
E0552023	[Message]	Illegal directive command is used.
	[Explanation]	An illegal instruction is entered.
	[Action by User]	Rewrite the instruction correctly.
E0552024	[Message]	No ';' at the top of comment.
	[Explanation]	';' is not entered at the beginning of a comment.
	[Action by User]	Enter a semicolon at the beginning of each comment. Check whether the mnemonic or operand is written correctly.
E0552026	[Message]	'CODE' section in big endian is not appropriate.
	[Explanation]	The value specified for the start address of the absolute-addressing CODE section is not a multiple of 4 while endian=big is specified.
	[Action by User]	Specify a multiple of 4 for the start address.
E0552027	[Message]	Illegal character code.
	[Explanation]	An illegal character code is specified.
E0552028	[Message]	Unrecognized character escape sequence.
	[Explanation]	An unrecognizable escape sequence is specified.
E0552029	[Message]	Invalid description in #pragma inline_asm function.
	[Explanation]	Invalid assembly-language code was usedin an assembly-language function. Go through the C-language source file and check the code corresponding to functions for which #pragma_inline_asm was specified.
E0552040	[Message]	Include nesting over.
	[Explanation]	Include is nested too many levels.
	[Action by User]	Rewrite include so that it is nested within 30 levels.
E0552041	[Message]	Can't open include file 'XXXX'.
	[Explanation]	The include file cannot be opened.
	[Action by User]	Check the include file name. Check the directory where the include file is stored.
E0552042	[Message]	Including the include file in itself.
	[Explanation]	An attempt is made to include the include file in itself.
	[Action by User]	Check the include file name and rewrite correctly.

E0552049	[Message]	Invalid reserved word exist in operand.
	[Explanation]	The operand contains a reserved word.
	[Action by User]	Reserved words cannot be written in an operand. Rewrite the operand correctly.
E0552050	[Message]	Operand value is not defined.
	[Explanation]	An undefined operand value is entered.
	[Action by User]	Write a valid value for operands.
E0552051	[Message]	'{' is missing.
	[Explanation]	'{' is not specified.
E0552052	[Message]	Addressing mode specifier is not appropriate.
	[Explanation]	The addressing mode specifier is written incorrectly.
	[Action by User]	Make sure that the addressing mode is written correctly.
E0552053	[Message]	Reserved word is missing.
	[Explanation]	No reserved word is entered.
E0552054	[Message]	']' is missing.
	[Explanation]	']' is not entered.
	[Action by User]	Write the right bracket ']' corresponding to the '['.
E0552055	[Message]	Right quote is missing.
	[Explanation]	A right quote is not entered.
	[Action by User]	Enter the right quote.
E0552056	[Message]	The value is not constant.
	[Explanation]	The value is indeterminate when assembled.
	[Action by User]	Write an expression, symbol name, or label name that will have a determinate value when assembled.
E0552057	[Message]	Quote is missing.
	[Explanation]	Quotes for a character string are not entered.
	[Action by User]	Enclose a character string with quotes as you write it.
E0552058	[Message]	Illegal operand is used.
	[Explanation]	The operand is incorrect.
	[Action by User]	Check the syntax for this operand and rewrite it correctly.
E0552059	[Message]	Operand number is not enough.
	[Explanation]	The number of operands is insufficient.
	[Action by User]	Check the syntax for these operands and rewrite them correctly.
E0552060	[Message]	Too many macro nesting.
	[Explanation]	The macro is nested too many levels.
	[Action by User]	Make sure that the macro is nested no more than 65,535 levels. Check the syntax for this source statement and rewrite it correctly.

E0552061	[Message]	Too many macro local label definition.
	[Explanation]	Too many macro local labels are defined.
	[Action by User]	Make sure that the number of macro local labels defined in one file are 65,535 or less.
E0552062	[Message]	'.MACRO' is missing for '.ENDM'.
	[Explanation]	.MACRO for .ENDM is not found.
	[Action by User]	Check the position where .ENDM is written.
E0552063	[Message]	'.MREPEAT' is missing for '.ENDR'.
	[Explanation]	.MREPEAT for .ENDR is not found.
	[Action by User]	Check the position where .ENDR is written.
E0552064	[Message]	'.MACRO' or '.MREPEAT' is missing for '.EXITM'.
	[Explanation]	.MACRO or .MREPEAT for .EXITM is not found.
	[Action by User]	Check the position where .EXITM is written.
E0552065	[Message]	No macro name.
	[Explanation]	No macro name is entered.
	[Action by User]	Write a macro name for each macro definition.
E0552066	[Message]	Too many formal parameter.
	[Explanation]	There are too many formal parameters defined for the macro.
	[Action by User]	Make sure that the number of formal parameters defined for the macro is 80 or less.
E0552067	[Message]	Illegal macro parameter.
	[Explanation]	The macro parameter contains some incorrect description.
	[Action by User]	Check the written contents of the macro parameter.
E0552068	[Message]	Source line is too long.
	[Explanation]	The source line is excessively long.
	[Action by User]	Check the contents written in the source line and correct it as necessary.
E0552069	[Message]	'.MACRO' is missing for '.LOCAL'.
	[Explanation]	.MACRO for .LOCAL is not found.
	[Action by User]	Check the position where .LOCAL is written. .LOCAL can only be written in a macro block.
E0552070	[Message]	No '.ENDM' statement.
	[Explanation]	.ENDM is not entered.
	[Action by User]	Check the position where .ENDM is written. Write .ENDM as necessary.
E0552071	[Message]	No '.ENDR' statement.
	[Explanation]	.ENDR is not entered.
	[Action by User]	Check the position where .ENDR is written. Write .ENDR as necessary.

E0552072	[Message]	')' is missing.
	[Explanation]	')' is not entered.
	[Action by User]	Write the right parenthesis ')' corresponding to the '('.
E0552073	[Message]	Operand expression is not completed.
	[Explanation]	The operand description is not complete.
	[Action by User]	Check the syntax for this operand and rewrite it correctly.
E0552074	[Message]	Syntax error in expression.
	[Explanation]	The expression is written incorrectly.
	[Action by User]	Check the syntax for this expression and rewrite it correctly.
E0552075	[Message]	String value exist in expression.
	[Explanation]	A character string is entered in the expression.
	[Action by User]	Rewrite the expression correctly.
E0552076	[Message]	Division by zero.
	[Explanation]	A divide by 0 operation is attempted.
	[Action by User]	Rewrite the expression correctly.
E0552077	[Message]	No '.END' statement.
	[Explanation]	.END is not entered.
	[Action by User]	Be sure to enter .END in the last line of the source program.
E0552078	[Message]	The specified address overlaps at '*address*'.
	[Explanation]	Something has already beenallocated to '*address*'.
	[Action by User]	Check the specifications for .ORG and .OFFSET.
E0552080	[Message]	'.IF' is missing for '.ELSE'.
	[Explanation]	.IF for .ELSE is not found.
	[Action by User]	Check the position where .ELSE is written.
E0552081	[Message]	'.IF' is missing for '.ELIF'.
	[Explanation]	.IF for .ELIF is not found.
	[Action by User]	Check the position where .ELIF is written.
E0552082	[Message]	'.IF' is missing for '.ENDIF'.
	[Explanation]	.IF for .ENDIF is not found.
	[Action by User]	Check the position where .ENDIF is written.
E0552083	[Message]	Too many nesting level of condition assemble.
	[Explanation]	Condition assembling is nested too many levels.
	[Action by User]	Check the syntax for this condition assemble statement and rewrite it correctly.
E0552084	[Message]	No '.ENDIF' statement.
	[Explanation]	No corresponding .ENDIF is found for the .IF statement in the source file.
	[Action by User]	Check the source description.

E0552088	[Message]	Can't open '.ASSERT' message file 'XXXX'.
	[Explanation]	.The .ASSERT output file cannot be opened.
	[Action by User]	Check the file name.
E0552089	[Message]	Can't write '.ASSERT' message file 'XXXX'.
	[Explanation]	Data cannot be written to the .ASSERT output file.
	[Action by User]	Check the permission of the file.
E0552090	[Message]	Too many temporary label.
	[Explanation]	There are too many temporary labels.
	[Action by User]	Replace the temporary labels with label names.
E0552091	[Message]	Temporary label is undefined.
	[Explanation]	The temporary label is not defined yet.
	[Action by User]	Define the temporary label.
E0552100	[Message]	Value is out of range.
	[Explanation]	The value is out of range.
	[Action by User]	Write a value that matches the register bit length.
E0552112	[Message]	Symbol is missing.
	[Explanation]	Symbol is not entered.
	[Action by User]	Write a symbol name.
E0552113	[Message]	Symbol definition is not appropriate.
	[Explanation]	The symbol is defined incorrectly.
	[Action by User]	Check the method for defining this symbol and rewrite it correctly.
E0552114	[Message]	Symbol has already defined as another type.
	[Explanation]	The symbol has already been defined in a different directive with the same name.
	[Action by User]	Change the symbol name.
E0552115	[Message]	Symbol has already defined as the same type.
	[Explanation]	The symbol has already been defined.
	[Action by User]	Change the symbol name.
E0552116	[Message]	Symbol is multiple defined.
	[Explanation]	The symbol is defined twice or more. The macro name and some other name are duplicates.
	[Action by User]	Change the symbol name.
E0552117	[Message]	Invalid label definition.
	[Explanation]	An invalid label is entered.
	[Action by User]	Rewrite the label definition.
E0552118	[Message]	Invalid symbol definition.
	[Explanation]	An invalid symbol is entered.
	[Action by User]	Rewrite the symbol definition.

E0552119	[Message]	Reserved word is used as label or symbol.
	[Explanation]	Reserved word is used as a label or symbol.
	[Action by User]	Rewrite the label or symbol name correctly.
E0552120	[Message]	Created symbol is too long
	[Explanation]	The label for a reserved word created by the **-create_unfilled_area** option is too long.
	[Action by User]	Shorten the file or section name.
E0552130	[Message]	No '.SECTION' statement.
	[Explanation]	.SECTION is not entered.
	[Action by User]	Always make sure that the source program contains at least one .SECTION.
E0552131	[Message]	Section type is not appropriate.
	[Action by User]	An instruction or a directive used in a section does not match the section type.
E0552132	[Message]	Section has already determined as attribute.
	[Explanation]	The attribute of this section has already been defined as relative. Directive command .ORG cannot be written here.
	[Action by User]	Check the attribute of the section.
E0552133	[Message]	Section attribute is not defined.
	[Explanation]	Section attribute is not defined. Directive command .ALIGN cannot be written in this section.
	[Action by User]	Make sure that directive .ALIGN is written in an absolute attribute section or a relative attribute section where ALIGN is specified.
E0552134	[Message]	Section name is missing.
	[Explanation]	No section name is entered.
	[Action by User]	Write a section name in the operand.
E0552135	[Message]	'ALIGN' is multiple specified in '.SECTION'.
	[Explanation]	Two or more ALIGN's are specified in the .SECTION definition line.
	[Action by User]	Delete extra ALIGN specifications.
E0552136	[Message]	Section type is multiple specified.
	[Explanation]	Section type is specified two or more times in the section definition line.
	[Action by User]	Only one section type CODE, DATA, or ROMDATA can be specified in a section definition line.
E0552137	[Message]	Too many operand.
	[Explanation]	There are extra operands.
	[Action by User]	Check the syntax for these operands and rewrite them correctly.
E0562000	[Message]	Invalid option : "*option*"
	[Explanation]	*option* is not supported.

E0562001	[Message]	Option "*option*" cannot be specified on command line
	[Explanation]	*option* cannot be specified on the command line.
	[Explanation]	Specify this option in a subcommand file.
E0562002	[Message]	Input option cannot be specified on command line
	[Explanation]	The input option was specified on the command line.
	[Action by User]	Input file specification on the command line should be made without the input option.
E0562003	[Message]	Subcommand option cannot be specified in subcommand file
	[Explanation]	The -subcommand option was specified in a subcommand file. The -subcommand option cannot be nested.
E0562004	[Message]	Option "*option1*" cannot be combined with option "*option2*"
	[Explanation]	*option 1* and *option 2* cannot be specified simultaneously.
E0562005	[Message]	Option "*option*" cannot be specified while processing "*process*"
	[Explanation]	*option* cannot be specified for *process*.
E0562006	[Message]	Option "*option1*" is ineffective without option "*option2*"
	[Explanation]	*option 1* requires option 2 be specified.
E0562010	[Message]	Option "*option*" requires parameter
	[Explanation]	*option* requires a parameter to be specified.
E0562011	[Message]	Invalid parameter specified in option "*option*" : "*parameter*"
	[Explanation]	An invalid parameter was specified for *option*.
E0562012	[Message]	Invalid number specified in option "*option*" : "*value*"
	[Explanation]	An invalid value was specified for *option*.
	[Action by User]	Check the range of valid values.
E0562013	[Message]	Invalid address value specified in option "*option*" : "*address*"
	[Explanation]	The address *address* specified in *option* is invalid.
	[Action by User]	A hexadecimal address between 0 and FFFFFFFF should be specified.
E0562014	[Message]	Illegal symbol/section name specified in "*option*" : "*name*"
	[Explanation]	The section or symbol name specified in *option* uses an illegal character.
E0562016	[Message]	Invalid alignment value specified in option "*option*" : "*alignment value*"
	[Explanation]	The alignment value specified in *option* is invalid.
	[Action by User]	1, 2, 4, 8, 16, or 32 should be specified.
E0562017	[Message]	Cannot output "*section*" specified in option "*option*"
	[Explanation]	Could not output a portion of the code in "section" specified by "option." Part of the instruction code in "section" has been swapped with instruction code in another section due to endian conversion.
	[Action by User]	Check the section address range with respect to 4-byte boundaries in the linkage list and find which section code is swapped with the target section code.

E0562020	[Message]	Duplicate file specified in option "*option*" : "*file*"
	[Explanation]	The same file was specified twice in *option*.
E0562022	[Message]	Address ranges overlap in option "*option*" : "*address range*"
	[Explanation]	Address ranges *address range* specified in *option* overlap.
E0562100	[Message]	Invalid address specified in cpu option : "*address*"
	[Explanation]	An address was specified with the -cpu option that cannot be specified for a cpu.
E0562101	[Message]	Invalid address specified in option "*option*" : "*address*"
	[Explanation]	The *address* specified in *option* exceeds the address range that can be specified by the cpu or the range specified by the cpu option.
E0562110	[Message]	Section size of second parameter in rom option is not 0 : "*section*"
	[Explanation]	The second parameter in the -rom option specifies "*section*" with non-zero size.
E0562111	[Message]	Absolute section cannot be specified in "*option*" option : "*section*"
	[Explanation]	An absolute address section was specified in *option*.
E0562114	[Message]	The generated duplicate section name "*section*" is confused
	[Explanation]	A section with the same name *section* appeared more than once and could not be processed.
E0562120	[Message]	Library "*file*" without module name specified as input file
	[Explanation]	A library file without a module name was specified as the input file.
E0562121	[Message]	Input file is not library file : "*file*(*module*)"
	[Explanation]	The file specified by *file* (*module*) as the input file is not a library file.
E0562130	[Message]	Cannot find file specified in option "*option*" : "*file*"
	[Explanation]	The file specified in *option* could not be found.
E0562131	[Message]	Cannot find module specified in option "*option*" : "*module*"
	[Explanation]	The module specified in *option* could not be found.
E0562132	[Message]	Cannot find "*name*" specified in option "*option*"
	[Explanation]	The symbol or section specified in *option* does not exist.
E0562133	[Message]	Cannot find defined symbol "*name*" in option "*option*"
	[Explanation]	The externally defined symbol specified in *option* does not exist.
E0562140	[Message]	Symbol/section "*name*" redefined in option "*option*"
	[Explanation]	The symbol or section specified in *option* has already been defined.
E0562141	[Message]	Module "*module*" redefined in option "*option*"
	[Explanation]	The module specified in *option* has already been defined.
E0562142	[Message]	Interrupt number "*vector number*" of "*section*" has multiple definition
	[Explanation]	Vector number definition was made multiple times in vector table *section*. Only one address can be specified for a vector number.
	[Action by User]	Check and correct the code in the source file.

E0562200	[Message]	Illegal object file : "*file*"
	[Explanation]	A format other than ELF format was input.
E0562201	[Message]	Illegal library file : "*file*"
	[Explanation]	*file* is not a library file.
E0562210	[Message]	Invalid input file type specified for option "*option*" : "*file*(*type*)"
	[Explanation]	When specifying *option*, a *file* (*type*) that cannot be processed was input.
E0562211	[Message]	Invalid input file type specified while processing "*process*" : "*file*(*type*)"
	[Explanation]	A *file* (*type*) that cannot be processed was input during processing *process*.
E0562212	[Message]	"*option*" cannot be specified for inter-module optimization information in "*file*"
	[Explanation]	The option *option* cannot be used because *file* includes inter-module optimization information.
	[Action by User]	Do not specify the goptimize option at compilation or assembly.
E0562220	[Message]	Illegal mode type "*mode type*" in "*file*"
	[Explanation]	A file with a different *mode type* was input.
E0562221	[Message]	Section type mismatch : "*section*"
	[Explanation]	Sections with the same name but different attributes (whether initial values present or not) were input.
E0562300	[Message]	Duplicate symbol "*symbol*" in "*file*"
	[Explanation]	There are duplicate occurrences of *symbol*.
E0562301	[Message]	Duplicate module "*module*" in "*file*"
	[Explanation]	There are duplicate occurrences of *module*.
E0562310	[Message]	Undefined external symbol "*symbol*" referenced in "*file*"
	[Explanation]	An undefined symbol *symbol* was referenced in *file*.
E0562311	[Message]	Section "*section1*" cannot refer to overlaid section : "*section2-symbol*"
	[Explanation]	A symbol defined in *section 1* was referenced in *section 2* that is allocated to the same address as *section 1* overlaid.
	[Action by User]	*section 1* and *section 2* must not be allocated to the same address.
E0562320	[Message]	Section address overflowed out of range : "*section*"
	[Explanation]	The address of *section* exceeds the usable address range.
E0562321	[Message]	Section "*section1*" overlaps section "*section2*"
	[Explanation]	The addresses of *section 1* and *section 2* overlap.
	[Action by User]	Change the address specified by the start option.
E0562330	[Message]	Relocation size overflow : "*file*"-"*section*"-"*offset*
	[Explanation]	The result of the relocation operation exceeded the relocation size. Possible causes include inaccessibility of a branch destination, and referencing of a symbol which must be located at a specific address.
	[Action by User]	Ensure that the referenced symbol at the offset position of section in the source list is placed at the correct position.

E0562331	[Message]	Division by zero in relocation value calculation : "*file*"-"*section*"-"*offset*"
	[Explanation]	Division by zero occurred during a relocation operation.
	[Action by User]	Check for problems in calculation of the position at *offset* in *section* in the source list.
E0562332	[Message]	Relocation value is odd number : "*file*"-"*section*"-"*offset*"
	[Explanation]	The result of the relocation operation is an odd number.
	[Action by User]	Check for problems in calculation of the position at *offset* in *section* in the source list.
E0562340	[Message]	Symbol name "*file*"-"*section*"-"*symbol*" is too long
	[Explanation]	The length of "*symbol*" in "*section*" exceeds the assembler translation limit.
	[Action by User]	To output a symbol address file, use a symbol name that is no longer than the assembler translation limit.
E0562366	[Message]	"*section*" specified in option "*option*" was moved other area.
E0562403	[Message]	Fast interrupt register in "*file*" conflicts with that in another file
	[Explanation]	The register number specified for the fast interrupt general register in *file* does not match the settings in other files.
	[Action by User]	Correct the register number to match the other settings and recompile the code.
E0562404	[Message]	Base register "*base register type*" in "*file*" conflicts with that in another file
	[Explanation]	The register number specified for *base register type* in *file* does not match the settings in other files.
	[Action by User]	Correct the register number to match the other settings and recompile the code.
E0562405	[Message]	Option "*compile option*" in "*file*" conflicts with that in another files
	[Explanation]	Specification of *compile option* is inconsistent between the input files.
	[Action by User]	Review the compile option.
E0562410	[Message]	Address value specified by map file differs from one after linkage as to "*symbol*"
	[Explanation]	The address of *symbol* differs between the address within the external symbol allocation information file used at compilation and the address after linkage.
	[Action by User]	Check (1) to (2) below. (1) Do not change the program before or after the map option specification at compilation. (2) rlink optimization may cause the sequence of the symbols after the map option specification at compilation to differ from that before the map option. Disable the map option at compilation or disable the rlink option for optimization.
E0562411	[Message]	Map file in "*file*" conflicts with that in another file
	[Explanation]	Different external symbol allocation information files were used by the input files at compilation.
E0562412	[Message]	Cannot open file : "*file*"
	[Explanation]	*file* (external symbol allocation information file) cannot be opened.
	[Action by User]	Check whether the file name and access rights are correct.

E0562413	[Message]	Cannot close file : "*file*"
	[Explanation]	*file* (external symbol allocation information file) cannot be closed. There may be insufficient disk space.
E0562414	[Message]	Cannot read file : "*file*"
	[Explanation]	*file* (external symbol allocation information file) cannot be read. There may be insufficient disk space.
E0562415	[Message]	Illegal map file : "*file*"
	[Explanation]	*file* (external symbol allocation information file) has an illegal format.
	[Action by User]	Check whether the file name is correct.
E0562416	[Message]	Order of functions specified by map file differs from one after linkage as to "*function name*"
	[Explanation]	The sequences of a function *function name* and those of other functions are different between the information within the external symbol allocation information file used at compilation and the location after linkage. The address of static within the function may be different between the external symbol allocation information file and the result after linkage.
E0562417	[Message]	Map file is not the newest version : "*file name*"
	[Explanation]	The external symbol allocation information file is not the latest version.
E0562420	[Message]	"*file1*" overlap address "*file2*" : "*address*"
	[Explanation]	The address specified for *file 1* is the same as that specified for *file 2*.
E0562600	[Message]	Library "*library*" requires "*licence edition*"
	[Explanation]	The "*library*" requires the "*edition*" edition.
E0563602	[Message]	"character string" option requires *Edition*.
E0572000	[Message]	Invalid option : "*option*"
	[Explanation]	"*option*" is not supported.
E0572200	[Message]	Illegal object file : "*file*"
	[Explanation]	The input file is not in the ELF format.
E0572500	[Message]	Cannot find library file : "*file*"
	[Explanation]	"*file*" specified as the library file was not found.
E0572501	[Message]	"*instance*" has been referenced as both an explicit specialization and a generated instantiation
	[Action by User]	For the file using "*instance*", confirm that form=relocate has not been used to generate a relocatable object file.
E0572502	[Message]	"*instance*" assigned to "*file1*" and "*file2*"
	[Explanation]	The definition of "*instance*" is duplicated in "*file1*" and "*file2*".
	[Action by User]	For the file using "*instance*", confirm that form=relocate has not been used to generate a relocatable object file.
E0573005	[Message]	Instantiation loop
	[Explanation]	An input file name may coincide with another file.
	[Action by User]	Change the filenames so that they do not coincide without the extension.

E0573007	[Message]	Cannot create instantiation request file "*file*"
	[Explanation]	The intermediate file for instantiation was not created.
	[Action by User]	Check whether access rights for the object creation directory are correct.
E0573008	[Message]	Cannot change to directory "*folder*"
	[Action by User]	Check if "*folder*" exists.
E0573009	[Message]	File "*file*" is read-only
	[Action by User]	Change the access rights.
E0573300	[Message]	Cannot open file : "*file*"
	[Action by User]	Check the filename and access rights.
E0573303	[Message]	Cannot read file : "*file*"
	[Explanation]	The input file was blank or there was not enough disk space.
E0573310	[Message]	Cannot open temporary file
	[Explanation]	An intermediate file cannot be opened. The HLNK_TMP specification was incorrect or there was not enough disk space.
E0573320	[Message]	Memory overflow
	[Explanation]	There is no more space in the usable memory within the linkage editor.
	[Action by User]	Increase the amount of memory available.
E0592001	[Message]	Multiple input files are not allowed.
	[Action by User]	Use a list file to convert more than one file.
E0592002	[Message]	Multiple output files are not allowed.
	[Action by User]	Use a list file to convert more than one file.
E0592003	[Message]	List file is specified more than once.
	[Action by User]	Combine them into a single list file.
E0592004	[Message]	Invalid argument for the "*option*" option.
	[Action by User]	Check the argument.
E0592005	[Message]	The "*option*" option can not have an argument.
	[Explanation]	An invalid argument was specified for the "*option*" option.
E0592006	[Message]	The "*option*" option requires an argument.
	[Explanation]	A required argument is missing from the "*option*" option specification.
E0592007	[Message]	The "*option*" option is specified more than once.
	[Explanation]	Option "*option*" can only be specified once at a time.
E0592008	[Message]	Requires an output file.
	[Explanation]	No output file has been specified for the specified input file.
E0592010	[Message]	Failed to open an output file "*file*".
E0592013	[Message]	Failed to delete a temporary file "*file*".
E0592015	[Message]	Failed to close an input file "*file*".
E0592016	[Message]	Failed to write an output file "*file*".

E0592018	[Message]	Failed to open an list file "*file*".
	[Action by User]	Make sure that the list file exists and has been specified correctly.
E0592019	[Message]	Syntax error in list file "*file*".
	[Explanation]	There is a syntax error in list file "*file*".
E0592020	[Message]	Failed to read a list file "*file*".
E0592101	[Message]	Unknown character '*C*'.
	[Explanation]	The pre-conversion C-language source file could not be converted, because it contains a character that is not permitted by the C language.
	[Action by User]	Edit the C-language source file and correct any syntax errors.
E0592102	[Message]	Illegal syntax in *string*.
	[Explanation]	The pre-conversion C-language source file could not be converted, because it contains a syntax error.
	[Action by User]	Edit the C-language source file and correct any syntax errors.
E0592201	[Message]	Illegal syntax.
	[Explanation]	The pre-conversion assembly-language source file could not be converted, because it contains a syntax error.
	[Action by User]	Edit the assembly-language source file and correct any syntax errors.
E0593002	[Message]	"-Xsfg_size_tidata_byte" size larger than "-Xsfg_size_tidata" size.
	[Action by User]	Set size "-Xsfg_size_tidata_byte" to equal to or less than size "-Xsfg_size_tidata", or size "-Xsfg_size_tidata" to greater than or equal to size "-Xsfg_size_tidata_byte".
E0593003	[Message]	Can not Read Symbol Information.
	[Explanation]	The symbol could not be loaded because there is no symbol-analysis information in memory, or it is corrupt.
	[Action by User]	Check the CX options and rebuild.
E0593004	[Message]	Can not Write the SFG file.
	[Explanation]	There could be a problem with disk space or user privileges.
	[Action by User]	Make sure that there is enough space to write the data, and check the user privileges.
E0594000	[Message]	Cannot find project file(*file name*).
	[Explanation]	There is no project file.
	[Action by User]	Make sure that the file exists.
E0594001	[Message]	Project file read error(*file name*).
	[Explanation]	An error occurred while loading the project file. Reading the project file may be blocked.
E0594002	[Message]	Illegal format in project file(*file name*).
	[Explanation]	The project file format is invalid.
	[Action by User]	This error occurs when invalid format is found in the project file. Either correct the error, or create the project again.

10.5.3 Fatal Errors

Table 10.4 Fatal Errors

F0511128	[Message]	Library file "*file-name*" is not found.
F0512003	[Message]	Too many errors.
F0520003	[Message]	#include file "*file*" includes itself.
	[Explanation]	#include file "*file*" includes itself. Correct the error.
F0520004	[Message]	Out of memory.
	[Action by User]	Out of memory. Close other applications, and perform the compile again.
F0520005	[Message]	Could not open source file "*file*".
F0520013	[Message]	Expected a file name.
F0520016	[Message]	"*file*" is not a valid source file name.
F0520035	[Message]	#error directive: *character string*
	[Explanation]	There is an "#error" directive in the source file.
F0520143	[Message]	Program too large or complicated to compile.
F0520163	[Message]	Could not open temporary file *xxx.*
F0520164	[Message]	Name of directory for temporary files is too long (*xxx*).
F0520182	[Message]	Could not open source file *xxx* (no directories in search list).
F0520189	[Message]	Error while writing "*file*" file.
F0520190	[Message]	Invalid intermediate language file.
F0520219	[Message]	Error while deleting file "*file*".
F0520542	[Message]	Could not create instantiation request file *name.*
F0520563	[Message]	Invalid preprocessor output file.
F0520564	[Message]	Cannot open preprocessor output file.
F0520571	[Message]	Invalid option: *option*
F0520641	[Message]	*xxx* is not a valid directory.
F0520642	[Message]	Cannot build temporary file name.
F0520869	[Message]	Could not set locale *xxx* to allow processing of multibyte characters.
F0520919	[Message]	Invalid output file: *xxx*
F0520920	[Message]	Cannot open output file: *xxx*
F0520926	[Message]	Cannot open definition list file: *xxx*
F0521083	[Message]	Exported template file *xxx* is corrupted.
F0521151	[Message]	Mangled name is too long.
F0521335	[Message]	Cannot open predefined macro file: *xxx*
F0521336	[Message]	Invalid predefined macro entry at line *line*: *line2*
F0521337	[Message]	Invalid macro mode name *xxx.*
F0521338	[Message]	Incompatible redefinition of predefined macro *xxx*.

F0523029	[Message]	Cannot open rule file
	[Explanation]	The file specified in the -misra2004="file name" or -misra2012="file name" option cannot be opened.
F0523030	[Message]	Incorrect description "*file name*" in rule file
	[Explanation]	The file specified in the -misra2004="*file name*" or -misra2012="file name" option includes illegal code.
F0523031	[Message]	Rule *rule number* is unsupported
	[Explanation]	The number of a rule that is not supported was specified.
F0523073	[Message]	"*instruction-set architecture name*" does not support this intrinsic function.
	[Explanation]	The specified intrinsic function is not supported. Check the **-isa** option setting.
F0523088	[Message]	Bit position is out of range.
F0523129	[Message]	The "*option name*" option is necessary for use of "*function*".
	[Explanation]	The option must be specified to use this function.
F0523300	[Message]	Cannot open internal file.
	[Explanation]	An intermediate file internally generated by the compiler cannot be opened.
F0523301	[Message]	Cannot close internal file.
	[Explanation]	An intermediate file internally generated by the compiler cannot be closed.
F0523302	[Message]	Cannot write internal file.
	[Explanation]	An error occurred while an intermediate file was being written to.
F0530320	[Message]	Duplicate symbol "*symbol name*".
F0530800	[Message]	Type of symbol "*symbol-name*" differs between files.
F0530808	[Message]	Alignment of variable "*variable-name*" differs between files.
F0530810	[Message]	#pragma directive for symbol "*symbol-name*" differs between files.
F0533021	[Message]	Out of memory.
	[Explanation]	Memory is insufficient.
	[Action by User]	Close other applications and recompile the program.
F0533300	[Message]	Cannot open an intermediate file.
	[Explanation]	A temporary file that was internally generated by the compiler cannot be opened.
F0533301	[Message]	Cannot close an intermediate file.
	[Explanation]	A temporary file that was internally generated by the compiler cannot be closed.
F0533302	[Message]	Cannot read an intermediate file.
	[Explanation]	An error occurred during reading of a temporary file.
F0533303	[Message]	Cannot write to an intermediate file.
	[Explanation]	An error occurred during writing of a temporary file.
F0533306	[Message]	Compilation was interrupted.
	[Explanation]	During compilation, an interrupt due to entry of the Cntl + C key combination was detected.

F0533330	[Message]	Cannot open an intermediate file.
	[Explanation]	A temporary file that was internally generated by the compiler cannot be opened.
F0540027	[Message]	Cannot read file "*file-name*".
F0540204	[Message]	Illegal stack access.
	[Explanation]	Attempted usage of the stack by a function has exceeded 2 Gbytes.
F0540300	[Message]	Cannot open an intermediate file.
	[Explanation]	A temporary file that was internally generated by the compiler cannot be opened.
F0540301	[Message]	Cannot close an intermediate file.
	[Explanation]	A temporary file that was internally generated by the compiler cannot be closed.
F0540302	[Message]	Cannot read an intermediate file.
	[Explanation]	An error occurred during reading of a temporary file.
F0540303	[Message]	Cannot write to an intermediate file.
	[Explanation]	An error occurred during writing of a temporary file.
F0540400	[Message]	Different parameters are set for the same #pramga "*identifier*".
F0544302	[Message]	Cannot read an intermediate file.
	[Explanation]	An error occurred while an intermediate file was being read.
F0544802	[Message]	The value of the parameter for the in-line function is outside the defined range.
	[Explanation]	The value of the parameter for the inline function is outside the supported range.
F0553000	[Message]	Can't create file '*filename*'.
	[Explanation]	The *filename* file cannot be generated.
	[Action by User]	Check the directory capacity.
F0553001	[Message]	Can't open file '*filename*'.
	[Explanation]	The *filename* file cannot be opened.
	[Action by User]	Check the file name.
F0553002	[Message]	Can't write file '*filename*'.
	[Explanation]	The *filename* file cannot be written to.
	[Action by User]	Check the permission of the file.
F0553003	[Message]	Can't read file '*filename*'.
	[Explanation]	The *filename* file cannot be read.
	[Action by User]	Check the permission of the file.
F0553004	[Message]	Can't create Temporary file.
	[Explanation]	Temporary file cannot be generated.
	[Action by User]	Specify a directory in environment variable TMP_RX so that a temporary file will be created in some place other than the current directory.
F0553005	[Message]	Can't open Temporary file.
	[Explanation]	The temporary file cannot be opened.
	[Action by User]	Check the directory specified in TMP_RX.

F0553006	[Message]	Can't read Temporary file.
	[Explanation]	The temporary file cannot be read.
	[Action by User]	Check the directory specified in TMP_RX.
F0553007	[Message]	Can't write Temporary file.
	[Explanation]	The temporary file cannot be written to.
	[Action by User]	Check the directory specified in TMP_RX.
F0553008	[Message]	Illegal file name '*filename*'.
	[Explanation]	The file name is illegal.
	[Action by User]	Specify a file name that conforms to file name description rules.
F0553016	[Message]	Lacking cpu specification.
	[Explanation]	No CPU type is specified.
	[Action by User]	Specify the CPU type by the cpuoption or environment variable CPU_RX.
F0553100	[Message]	Command line is too long.
	[Explanation]	The command line has too many characters.
	[Action by User]	Re-input the command.
F0553101	[Message]	Invalid option '*xx*' is used.
	[Explanation]	An invalid command option *xx* is used.
	[Action by User]	The specified option is nonexistent. Re-input the command correctly.
F0553102	[Message]	Ignore option '*xx*'.
	[Explanation]	An invalid option is specified.
F0553103	[Message]	Option '*xx*' is not appropriate.
	[Explanation]	Command option *xx* is written incorrectly.
	[Action by User]	Specify the command option correctly again.
F0553104	[Message]	No input files specified.
	[Explanation]	No input file is specified.
	[Action by User]	Specify an input file.
F0553105	[Message]	Source files number exceed 80.
	[Explanation]	The number of source files exceeds 80.
	[Action by User]	Execute assembling separately in two or more operations.
F0553106	[Message]	Lacking cpu specification.
	[Explanation]	No CPU type is specified.
	[Action by User]	Specify the CPU type by the cpuoption or environment variable CPU_RX.
F0553110	[Message]	Multiple register base/fint_register.
	[Explanation]	A single register is specified by the baseand fint_registeroptions.
F0553111	[Message]	Multiple register base/pid.
	[Explanation]	A single register is specified by the baseand pidoptions.

F0553112	[Message]	Multiple register base/nouse_pid_register.
	[Explanation]	A single register is specified by the baseand nouse_pid_registeroptions.
F0553113	[Message]	Neither isa nor cpu is specified
F0553114	[Message]	Both '-isa' option and '-cpu' option are specified
F0553115	[Message]	The '-cpu' option and the '-fpu' option are inconsistent
F0553200	[Message]	Error occurred in executing '*xxx*'.
	[Explanation]	An error occurred when executing *xxx*.
	[Action by User]	Review the environment variable settings.
F0553201	[Message]	Not enough memory.
	[Explanation]	Memory is insufficient.
	[Action by User]	Divide the file and re-run. Orincrease the memory capacity.
F0553202	[Message]	Can't find work dir.
	[Explanation]	The work directory is not found.
	[Action by User]	Make sure that the setting of environment variable TMP_RX is correct.
F0563000	[Message]	No input file
	[Explanation]	There is no input file.
F0563001	[Message]	No module in library
	[Explanation]	There are no modules in the library.
F0563002	[Message]	Option "*option1*" is ineffective without option "*option2*"
	[Explanation]	The option *option 1* requires that the option *option 2* be specified.
F0563003	[Message]	Illegal file format "*file*"
F0563004	[Message]	Invalid inter-module optimization information type in "*file*"
	[Explanation]	The file contains an unsupported inter-module optimization information *type*.
	[Action by User]	Check if the compiler and assembler versions are correct.
F0563020	[Message]	No cpu information in input files
	[Explanation]	The CPU type cannot be identified from the input file.
	[Action by User]	Check that the binary file is specified with the -binary option and the .obj or .rel files to be linked together exist.
F0563100	[Message]	Section address overflow out of range : "*section*"
	[Explanation]	The address of *section* exceeded the area available.
	[Action by User]	Change the address specified by the start option. For details of the address space, refer to the hardware manual of the target CPU.
F0563102	[Message]	Section contents overlap in absolute section "*section*" in "*file*"
	[Explanation]	Data addresses overlap within an absolute address section.
	[Action by User]	Modify the source program.
F0563103	[Message]	Section size overflow : "*section*"
	[Explanation]	Section "*section*" has exceeded the usable size.

F0563110	[Message]	Illegal cpu type "*cpu type*" in "*file*"
	[Explanation]	A file with a different cpu type was input.
F0563111	[Message]	Illegal encode type "*endian type*" in "*file*"
	[Explanation]	A file with a different endian type was input.
F0563112	[Message]	Invalid relocation type in "*file*"
	[Explanation]	There is an unsupported relocation type in *file*.
	[Action by User]	Ensure the compiler and assembler versions are correct.
F0563115	[Message]	Cpu type in "*file*" is not supported
	[Explanation]	The CPU type specified in "*file*" is not supported. Check if the input file is correct.
F0563120	[Message]	Illegal size of the absolute code section : "*section*" in "*file*"
	[Explanation]	Absolute-addressing section *section* in *file* has an illegal size.
F0563150	[Message]	Multiple files cannot be specified while processing "*process*"
F0563200	[Message]	Too many sections
	[Explanation]	The number of sections exceeded the translation limit. It may be possible to eliminate this problem by specifying multiple file output.
F0563201	[Message]	Too many symbols
	[Explanation]	The number of symbols exceeded the translation limit. It may be possible to eliminate this problem by specifying multiple file output.
F0563202	[Message]	Too many modules
	[Explanation]	The number of modules exceeded the translation limit.
	[Action by User]	Divide the library.
F0563203	[Message]	Reserved module name "rlink_generates"
	[Explanation]	rlink_generates_** (** is a value from 01 to 99) is a reserved name used by the optimizing linkage editor. It is used as an .obj or .rel file name or a module name within a library.
	[Action by User]	Modify the name if it is used as a file name or a module name within a library.
F0563204	[Message]	Reserved section name "$sss_fetch"
	[Explanation]	sss_fetch** (sss is any string, and ** is a value from 01 to 99) is a reserved name used by the optimizing linkage editor.
	[Action by User]	Change the symbol name or section name.
F0563300	[Message]	Cannot open file : "*file*"
	[Explanation]	*file* cannot be opened.
	[Action by User]	Check whether the file name and access rights are correct.
F0563301	[Message]	Cannot close file : "*file*"
	[Explanation]	*file* cannot be closed. There may be insufficient disk space.
F0563302	[Message]	Cannot write file : "*file*"
	[Explanation]	Writing to *file* is not possible. There may be insufficient disk space.

F0563303	[Message]	Cannot read file : "*file*"
	[Explanation]	*file* cannot be read. An empty file may have been input, or there may be insufficient disk space.
F0563310	[Message]	Cannot open temporary file
	[Explanation]	A temporary file cannot be opened.
	[Action by User]	Check to ensure the HLNK_TMP specification is correct, or there may be insufficient disk space.
F0563314	[Message]	Cannot delete temporary file
	[Explanation]	A temporary file cannot be deleted. There may be insufficient disk space.
F0563320	[Message]	Memory overflow
	[Explanation]	There is no more space in the usable memory within the linker.
	[Action by User]	Increase the amount of memory available.
F0563400	[Message]	Cannot execute "*load module*"
	[Explanation]	*load module* cannot be executed.
	[Action by User]	Check whether the path for *load module* is set correctly.
F0563410	[Message]	Interrupt by user
	[Explanation]	An interrupt generated by (Ctrl) + C keys from a standard input terminal was detected.
F0563420	[Message]	Error occurred in "*load module*"
	[Explanation]	An error occurred while executing *load module*.
F0563430	[Message]	The total section size exceeded the limit of the evaluation version of *version*. Please consider purchasing the product.
F0563431	[Message]	Incorrect device type, object file mismatch.
F0563600	[Message]	Option "*option*" requires parameter
F0563601	[Message]	Invalid parameter specified in option "*option*" : "*parameter*"
F0578200	[Message]	memory allocation fault
	[Explanation]	Not enough memory.
F0578201	[Message]	bad key *character* - use [dm(a|b)qr(a|b|u)txV]
	[Explanation]	*character* cannot be specified as a key.
F0578202	[Message]	bad option *character* - use [cv]
	[Explanation]	*character* cannot be specified as an option.
F0578203	[Message]	bad option *string*
	[Explanation]	*string* cannot be specified as an option.
F0578204	[Message]	can not create file *file*
	[Explanation]	Could not create file *file*.
F0578205	[Message]	file name *name*... is too long - limit is *number*
	[Explanation]	File name *name* is too long. The maximum value is *number1*.

F0578206	[Message]	can not open file *file*
	[Explanation]	Could not open file *file*.
F0578207	[Message]	can not close file *file*
	[Explanation]	Could not close file *file*.
F0578208	[Message]	can not read file *file*
	[Explanation]	Cannot read from file *file*.
F0578209	[Message]	can not write file *file*
	[Explanation]	Cannot write to file *file*.
F0578210	[Message]	can not seek file *file*
	[Explanation]	Cannot seek in file *file*.
F0578212	[Message]	can not nest command file *file*
	[Explanation]	Command file *file* is nested. Nesting is not allowed.
F0578213	[Message]	file is not library *file*
	[Explanation]	*file* is not a library file.
F0578214	[Message]	malformed library file *file*
	[Explanation]	Library file *file* could be corrupt.
F0578215	[Message]	can not find member *member*
	[Explanation]	Member *member* not found in library file.
F0578216	[Message]	symbol table limit error *file* (*number1*) - limit is *number2*
	[Explanation]	The number of symbols *number1* in library file *file* exceeds the maximum limit. The maximum value is *number2*.
F0578217	[Message]	symbol table error *file*
	[Explanation]	Failed to create a library file table for library file *file*.
F0578218	[Message]	string table error *file*
	[Explanation]	The library string table for library file *file* could be corrupt.
F0578219	[Message]	*file* has no member
	[Explanation]	There are no members in library file *file*.
F0578220	[Message]	version error *file*
	[Explanation]	The version of the format of the specified file *file* is not supported by this librarian.
F0578221	[Message]	can not read library header *file*
	[Explanation]	Cannot read header from library file *file*.
F0593021	[Message]	Memory overflow
	[Explanation]	Memory is insufficient. Close other applications, and generate the library again.
F0593113	[Message]	Neither isa nor cpu is specified
F0593114	[Message]	Both '-isa' option and '-cpu' option are specified
F0593300	[Message]	Cannot open internal file
	[Explanation]	The internal file cannot be opened.

F0593302	[Message]	Cannot input internal file
	[Explanation]	An attempt to read the internal file failed.
F0593303	[Message]	Cannot output internal file
	[Explanation]	An attempt to write to the internal file failed.
F0593305	[Message]	Invalid command parameter "*option-name*"
	[Explanation]	Invalid specification of "*option-name*"
F0593320	[Message]	Command parameter buffer overflow
	[Explanation]	Internal buffer is insufficient.
F0593321	[Message]	Illegal environment variable
	[Explanation]	Environment settings are specified incorrectly. Review the settings.
F0593322	[Message]	Lacking cpu specification
	[Explanation]	The -cpu option is not specified. Check the setting.
F0593324	[Message]	Cannot open subcommand file "*subcommand-file-name*"
	[Explanation]	"*subcommand-file-name*" cannot be opened.
F0593325	[Message]	Cannot close subcommand file
	[Explanation]	The subcommand file cannot be closed.
F0593326	[Message]	Cannot input subcommand file
	[Explanation]	An attempt to read the subcommand file failed.
F0593327	[Message]	Cannot get compiler version
	[Explanation]	The compiler version cannot be acquired.

10.5.4 Informations

Table 10.5 Informations

M0520009	[Message]	Nested comment is not allowed.
M0520018	[Message]	Expected a ")".
M0520111	[Message]	Statement is unreachable.
M0520128	[Message]	Loop is not reachable from preceding code.
M0520174	[Message]	Expression has no effect.
M0520177	[Message]	*Type* "*symbol*" was declared but never referenced.
M0520193	[Message]	Zero used for undefined preprocessing identifier *xxx*.
M0520237	[Message]	Selector expression is constant.
M0520261	[Message]	Access control not specified ("*name*" by default).
M0520324	[Message]	Duplicate friend declaration.
M0520381	[Message]	Extra ";" ignored.
M0520399	[Message]	*name* has an operator new *xxx*() but no default operator delete *xxx*().
M0520400	[Message]	*name* has a default operator delete *xxx*() but no operator new *xxx*().
M0520479	[Message]	*name redeclared* "inline" after being called.
M0520487	[Message]	Inline *name* cannot be explicitly instantiated.
M0520534	[Message]	Use of a local type to specify an exception.
M0520535	[Message]	Redundant type in exception specification.
M0520549	[Message]	*symbol* is used before its value is set.
M0520550	[Message]	*Type* "*symbol*" was set but never used.
M0520618	[Message]	Struct or union declares no named members.
M0520652	[Message]	Calling convention is ignored for this type.
M0520678	[Message]	Call of "*symbol*" cannot be inlined.
M0520679	[Message]	*symbol* cannot be inlined.
M0520815	[Message]	Type qualifier on return type is meaningless.
M0520826	[Message]	*name* was never referenced.
M0520831	[Message]	Support for placement delete is disabled.
M0520863	[Message]	Effect of this "#pragma pack" directive is local to *xxx*.
M0520866	[Message]	Exception specification ignored.
M0520949	[Message]	Specifying a default argument on this declaration is nonstandard.
M0521348	[Message]	Declaration hides "*symbol*".
M0521353	[Message]	*symbol* has no corresponding member operator delete *xxx* (to be called if an exception is thrown during initialization of an allocated object).
M0521380	[Message]	Virtual *xxx* was not defined (and cannot be defined elsewhere because it is a member of an unnamed namespace).

M0521381	[Message]	Carriage return character in source line outside of comment or character/string literal.
M0523009	[Message]	This pragma has no effect.
M0523028	[Message]	Rule *rule number* : *description*
	[Explanation]	Violation of a MISRA-C:2004 rule (indicated by the rule number and description) was detected.
M0523033	[Message]	Precision lost.
M0523086	[Message]	Rule *rule number* : *description*
	[Explanation]	Violation of a MISRA-C:2012 rule (indicated by the rule number and description) was detected.
M0560002	[Message]	Symbol "*symbol*" created by optimization "*optimization*"
	[Explanation]	The symbol named *symbol* was created as a result of the optimization.
M0560004	[Message]	"*file*"-"*symbol*" deleted by optimization
	[Explanation]	As a result of symbol_delete optimization, the symbol named *symbol* in *file* was deleted.
M0560005	[Message]	The offset value from the symbol location has been changed by optimization "*file*"-"*section*"-"*symbol* ± offset"
	[Explanation]	As a result of the size being changed by optimization within the range of symbol ± offset, the offset value was changed. Check that this does not cause a problem. To disable changing of the offset value, cancel the specification of the goptimize option on assembly of file.
M0560100	[Message]	No inter-module optimization information in "*file*"
	[Explanation]	No inter-module optimization information was found in *file*. Inter-module optimization is not performed on *file*. To perform inter-module optimization, specify the goptimize option on compiling and assembly.
M0560101	[Message]	No stack information in "*file"*
	[Explanation]	No stack information was found in *file*. *file* may be an assembler output file. The contents of the file will not be in the stack information file output by the linker.
M0560102	[Message]	Stack size "*size*" specified to the undefined symbol "*symbol*" in "*file*"
	[Explanation]	Stack size *size* is specified for the undefined symbol named symbol in *file*.
M0560103	[Message]	Multiple stack sizes specified to the symbol "*symbol*"
	[Explanation]	Multiple stack sizes are specified for the symbol named *symbol*.
M0560300	[Message]	Mode type "*mode type1*" in "*file*" differ from "*mde type2*"
	[Explanation]	A file with a different mode type was input.
M0560400	[Message]	Unused symbol "*file*"-"*symbol*"
	[Explanation]	The symbol named *symbol* in *file* is not used.
M0560500	[Message]	Generated CRC code at "*address*"
	[Explanation]	CRC code was generated at *address*.
M0560510	[Message]	Section "*section*" was moved other area specified in option "cpu=<*attribute*>"
	[Explanation]	*section* without dividing is allocated according to cpu=<*attribute*>.

M0560511	[Message]	Sections "*section name*","*new section name*" are Non-contiguous
	[Explanation]	*section* was divided and the newly created section is new section name.
M0560700	[Message]	Section address overflow out of range : "*section*"

10.5.5 Warnings

Table 10.6 Warnings

W0511105	[Message]	"*path*" specified by the "*character string*" option is a file. Specify a folder.
W0511106	[Message]	The folder "*folder*" specified by the "*character string*" option is not found.
W0511123	[Message]	The "*character string2*" option is ignored when the "*character string1*" option is specified at the same time.
W0511143	[Message]	The "-Xfloat" option is ignored because specified device does not have FPU.
W0511144	[Message]	"-C" option and "-Xcommon" option is mismatch. Instruction set by "*character string1*" option is ignored. Create common object for "*character string2*" instruction set.
W0511146	[Message]	"*symbol name*" specified in the "*character string*" option is not allowed for a preprocessor macro. Recognized only as an assembler symbol.
W0511147	[Message]	The "*character string*" option is specified more than once. The latter is valid.
W0511149	[Message]	The "*character string2*" option is ignored when the "*character string1*" option and the "*character string2*" option are inconsistent.
W0511151	[Message]	The "*character string2*" option is ignored when the "*character string1*" option is not specified.
W0511153	[Message]	Optimization itemoptions were cleared when "-*Ocharacter string*" option is specified. Optimization itemoptions need to specify after "-*Ocharacter string*" option.
W0511156	[Message]	Device file is not found in the folder specified by the "-Xdev_path" option.
	[Explanation]	Device file is not found in the folder specified by the "-Xdev_path" option. Will search standard device file folder.
W0511164	[Message]	Duplicate file name. "*file-name*".
	[Explanation]	The same file name "*file-name*" is specified more than once.
W0511166	[Message]	"*macro name*" is not a valid predefined macro name.
	[Explanation]	Specification of the undefine option is invalid.
W0511168	[Message]	"*option-name*" option has no effect in this version.
W0511169	[Message]	"*code*" is not valid in "*language specifications*"
	[Explanation]	"*code*" is invalid in the "language specifications (C or C++)".
W0511170	[Message]	"*option-name*" option is ignored due to the specification of another option.
W0511171	[Message]	"*code*" is ignored in "*language specifications*".
	[Explanation]	"*code*" is ignored in the "language specifications (C or C++)".
W0511172	[Message]	Nothing to compile, assemble or link.(input and output combination)
	[Action by User]	Check the combination of the input file and output option.
W0511179	[Message]	The evaluation version is valid for the remaining *number* days.
W0511180	[Message]	The evaluation period of *version* has expired.
W0511185	[Message]	The trial period for the features of the Professional edition expires in *number* days. Please consider purchasing the product of Professional edition.
W0519999	[Message]	The "*option-name*" option is not implemented.

W0520001	[Message]	Last line of file ends without a newline.
	[Action by User]	Add a line break.
W0520009	[Message]	Nested comment is not allowed.
	[Action by User]	Eliminate nesting.
W0520011	[Message]	Unrecognized preprocessing directive.
W0520012	[Message]	Parsing restarts here after previous syntax error.
W0520014	[Message]	Extra text after expected end of preprocessing directive.
W0520019	[Message]	Extra text after expected end of number.
W0520021	[Message]	Type qualifiers are meaningless in this declaration.
	[Explanation]	Type qualifiers are meaningless in this declaration. Ignored.
W0520026	[Message]	Too many characters in character constant.
	[Explanation]	Too many characters in character constant. Character constants cannot contain more than one character.
W0520027	[Message]	Character value is out of range.
W0520038	[Message]	Directive is not allowed -- an #else has already appeared.
	[Explanation]	Since there is a preceding #else, this directive is illegal.
W0520039	[Message]	Division by zero.
W0520042	[Message]	Operand types are incompatible ("*type1*" and "*type2*").
W0520045	[Message]	#undef may not be used on this predefined name.
W0520046	[Message]	*symbol* is predefined; attempted redefinition ignored.
W0520047	[Message]	Incompatible redefinition of macro "symbol".
W0520054	[Message]	Too few arguments in macro invocation.
W0520055	[Message]	Too many arguments in macro invocation.
W0520061	[Message]	Integer operation result is out of range.
W0520062	[Message]	Shift count is negative.
	[Explanation]	Shift count is negative. The behavior will be undefined in ANSI-C.
W0520063	[Message]	Shift count is too large.
W0520064	[Message]	Declaration does not declare anything.
W0520066	[Message]	Enumeration value is out of "int" range.
W0520068	[Message]	Integer conversion resulted in a change of sign.
W0520069	[Message]	Integer conversion resulted in truncation.
	[Explanation]	The conversion result of the integer type was truncated.
W0520070	[Message]	Incomplete type is not allowed.
W0520076	[Message]	Argument to macro is empty.
W0520077	[Message]	This declaration has no storage class or type specifier.
W0520082	[Message]	Storage class is not first.
	[Explanation]	Storage class is not first. Specify the declaration of the storage class first.

W0520083	[Message]	Type qualifier specified more than once.
W0520085	[Message]	Invalid storage class for a parameter.
W0520086	[Message]	Invalid storage class for a function.
W0520099	[Message]	A declaration here must declare a parameter.
W0520101	[Message]	*xxx* has already been declared in the current scope.
W0520108	[Message]	Signed bit field of length 1.
W0520111	[Message]	Statement is unreachable.
W0520117	[Message]	Non-void "*symbol*" should return a value.
W0520118	[Message]	A void function may not return a value.
W0520127	[Message]	Expected a statement.
W0520128	[Message]	Loop is not reachable from preceding code.
W0520138	[Message]	Taking the address of a register variable is not allowed.
W0520139	[Message]	Taking the address of a bit field is not allowed.
W0520140	[Message]	Too many arguments in function call.
W0520144	[Message]	A value of type "*type1*" cannot be used to initialize an entity of type "*type2*".
W0520147	[Message]	Declaration is incompatible with "*declaration*" (declared at line *number*).
W0520152	[Message]	Conversion of nonzero integer to pointer.
W0520157	[Message]	Expression must be an integral constant expression.
W0520161	[Message]	Unrecognized #pragma.
W0520165	[Message]	Too few arguments in function call.
W0520167	[Message]	Argument of type "*type1*" is incompatible with parameter of type "*type2*".
W0520170	[Message]	Pointer points outside of underlying object.
W0520171	[Message]	Invalid type conversion.
W0520172	[Message]	External/internal linkage conflict with previous declaration.
W0520173	[Message]	Floating-point value does not fit in required integral type.
W0520174	[Message]	Expression has no effect.
	[Explanation]	Expression has no effect. It is invalid.
W0520175	[Message]	Subscript out of range.
W0520177	[Message]	*Type* "*symbol*" was declared but never referenced.
W0520179	[Message]	Right operand of "%" is zero.
W0520180	[Message]	Argument is incompatible with formal parameter.
W0520181	[Message]	Argument is incompatible with corresponding format string conversion.
W0520185	[Message]	Dynamic initialization in unreachable code.
W0520186	[Message]	Pointless comparison of unsigned integer with zero.
W0520187	[Message]	Use of "=" where "==" may have been intended.
W0520188	[Message]	Enumerated type mixed with another type.

W0520191	[Message]	Type qualifier is meaningless on cast type.
W0520192	[Message]	Unrecognized character escape sequence.
W0520223	[Message]	Function *xxx* declared implicitly.
W0520224	[Message]	The format string requires additional arguments.
W0520225	[Message]	The format string ends before this argument.
W0520226	[Message]	Invalid format string conversion.
W0520229	[Message]	Bit field cannot contain all values of the enumerated type.
W0520231	[Message]	Declaration is not visible outside of function.
W0520232	[Message]	Old-fashioned typedef of "void" ignored.
W0520236	[Message]	Controlling expression is constant.
W0520257	[Message]	const "*symbol*" requires an initializer.
W0520260	[Message]	Explicit type is missing ("int" assumed).
W0520262	[Message]	Not a class or struct name.
W0520280	[Message]	Declaration of a member with the same name as its class.
W0520284	[Message]	NULL reference is not allowed.
W0520296	[Message]	Invalid use of non-lvalue array.
W0520300	[Message]	A pointer to a bound function may only be used to call the function.
W0520301	[Message]	typedef name has already been declared (with same type).
W0520326	[Message]	Inline is not allowed.
W0520335	[Message]	Linkage specification is not allowed.
W0520368	[Message]	*xxx* defines no constructor to initialize the following:
W0520370	[Message]	*symbol* has an uninitialized const field.
W0520375	[Message]	Declaration requires a typedef name.
W0520377	[Message]	"virtual" is not allowed.
W0520381	[Message]	Extra ";" ignored.
W0520382	[Message]	In-class initializer for nonstatic member is nonstandard.
W0520414	[Message]	Delete of pointer to incomplete class.
W0520430	[Message]	Returning reference to local temporary.
W0520494	[Message]	Declaring a void parameter list with a typedef is nonstandard.
W0520497	[Message]	Declaration of %sq hides template parameter.
W0520512	[Message]	Type qualifier on a reference type is not allowed.
W0520513	[Message]	A value of type "*type1*" cannot be assigned to an entity of type "*type2*".
W0520514	[Message]	Pointless comparison of unsigned integer with a negative constant.
W0520520	[Message]	Initialization with "{...}" expected for aggregate object.
W0520522	[Message]	Pointless friend declaration.
W0520523	[Message]	"." used in place of "::" to form a qualified name.

W0520533	[Message]	Handler is potentially masked by previous handler for type "type".
W0520541	[Message]	Omission of exception specification is incompatible with previous *name*.
W0520549	[Message]	*Type* "*symbol*" is used before its value is set.
W0520550	[Message]	*Type* "*symbol*" was set but never used.
W0520552	[Message]	Exception specification is not allowed.
W0520553	[Message]	external/internal linkage conflict for "symbol".
W0520554	[Message]	*name* will not be called for implicit or explicit conversions.
W0520611	[Message]	Overloaded virtual function *name1* is only partially overridden in *name2*.
W0520617	[Message]	Pointer-to-member-function cast to pointer to function.
W0520618	[Message]	struct or union declares no named members.
W0520650	[Message]	Calling convention specified here is ignored.
W0520657	[Message]	Inline specification is incompatible with previous "*symbol*".
W0520662	[Message]	Call of pure virtual function.
W0520676	[Message]	Using out-of-scope declaration of *type* "*symbol*" (declared at line *number*).
W0520691	[Message]	*xxx*, required for copy that was eliminated, is inaccessible.
W0520692	[Message]	*xxx*, required for copy that was eliminated, is not callable because reference parameter cannot be bound to rvalue.
W0520708	[Message]	Incrementing a bool value is deprecated.
W0520720	[Message]	Redeclaration of *xxx* is not allowed to alter its access.
W0520722	[Message]	Use of alternative token "<:" appears to be unintended.
W0520723	[Message]	Use of alternative token "%%:" appears to be unintended.
W0520737	[Message]	Using-declaration ignored -- it refers to the current namespace.
W0520748	[Message]	Calling convention specified more than once.
W0520760	[Message]	*symbol* explicitly instantiated more than once.
W0520767	[Message]	Conversion from pointer to smaller integer.
W0520780	[Message]	Reference is to *symbol1* -- under old for-init scoping rules it would have been *symbol2*.
W0520783	[Message]	Empty comment interpreted as token-pasting operator "##".
W0520794	[Message]	Template parameter %sq may not be used in an elaborated type specifier.
W0520802	[Message]	Specifying a default argument when redeclaring an unreferenced function template is nonstandard.
W0520806	[Message]	Omission of exception specification is incompatible with *name*.
W0520812	[Message]	const object requires an initializer -- class *type* has no explicitly declared default constructor.
W0520815	[Message]	Type qualifier on return type is meaningless.
W0520825	[Message]	Virtual inline *name* was never defined.
W0520826	[Message]	*name* was never referenced.
W0520829	[Message]	Double used for "long double" in generated C code.

W0520830	[Message]	*xxx* has no corresponding operator delete*yyy* (to be called if an exception is thrown during initialization of an allocated object).
W0520831	[Message]	Support for placement delete is disabled
W0520836	[Message]	Returning reference to local variable.
W0520837	[Message]	Omission of explicit type is nonstandard ("int" assumed).
W0520867	[Message]	Declaration of "size_t" does not match the expected type "*type*".
W0520870	[Message]	Invalid multibyte character sequence.
W0520902	[Message]	Type qualifier ignored.
W0520912	[Message]	Ambiguous class member reference -- *symbol1* used in preference to *symbol2*.
W0520925	[Message]	Type qualifiers on function types are ignored.
W0520936	[Message]	Redeclaration of *name* alters its access.
W0520940	[Message]	Missing return statement at end of non-void "*symbol*"..
W0520941	[Message]	Duplicate using-declaration of *name* ignored.
W0520942	[Message]	enum bit-fields are always unsigned, but enum %t includes negative enumerator.
W0520948	[Message]	Nonstandard local-class friend declaration -- no prior declaration in the enclosing scope.
W0520951	[Message]	Return type of function "main" must be "int".
W0520959	[Message]	Declared size for bit field is larger than the size of the bit field type; truncated to any-string bits.
W0520961	[Message]	Use of a type with no linkage to declare a variable with linkage.
W0520962	[Message]	Use of a type with no linkage to declare a function.
W0520970	[Message]	The qualifier on this friend declaration is ignored.
W0520973	[Message]	Inline used as a function qualifier is ignored.
W0520984	[Message]	operator new and operator delete cannot be given internal linkage.
W0520991	[Message]	Extra braces are nonstandard.
W0520993	[Message]	Subtraction of pointer types "*type name 1*" and "*type name 2*" is nonstandard.
W0520997	[Message]	*function2* is hidden by function*1* -- virtual function override intended?
W0521000	[Message]	A storage class may not be specified here.
W0521028	[Message]	Invalid redeclaration of nested class.
W0521030	[Message]	A variable with static storage duration cannot be defined within an inline function.
W0521046	[Message]	Floating-point value cannot be represented exactly.
W0521050	[Message]	Imaginary *= imaginary sets the left-hand operand to zero.
W0521051	[Message]	Standard requires that "*symbol*" be given a type by a subsequent declaration ("int" assumed).
W0521053	[Message]	Conversion from integer to smaller pointer.
W0521055	[Message]	Types cannot be declared in anonymous unions.
W0521056	[Message]	Returning pointer to local variable.
W0521057	[Message]	Returning pointer to local temporary.

W0521072	[Message]	A declaration cannot have a label.
W0521105	[Message]	#warning directive: *character string*.
	[Explanation]	There is a "#warning" directive in the source file.
W0521145	[Message]	*type1* would have been promoted to "*type2*" when passed through the ellipsis parameter; use the latter type instead.
W0521163	[Message]	va_start should only appear in a function with an ellipsis parameter.
W0521192	[Message]	Null (zero) character in input line ignored.
W0521193	[Message]	Null (zero) character in string or character constant.
W0521194	[Message]	Null (zero) character in header name.
W0521197	[Message]	The prototype declaration of %nfd is ignored after this unprototyped redeclaration.
W0521211	[Message]	Nonstandard cast to array type ignored.
W0521213	[Message]	field uses tail padding of a base class.
W0521218	[Message]	Base class *xxx* uses tail padding of base class *yyy*.
W0521222	[Message]	Invalid error number.
W0521223	[Message]	Invalid error tag.
W0521224	[Message]	Expected an error number or error tag.
W0521235	[Message]	Nonstandard conversion between pointer to function and pointer to data.
W0521273	[Message]	Alignment-of operator applied to incomplete type.
W0521285	[Message]	Nonstandard qualified name in namespace member declaration.
W0521290	[Message]	Non-POD class type passed through ellipsis.
W0521294	[Message]	Integer operand may cause fixed-point overflow.
W0521296	[Message]	Fixed-point value cannot be represented exactly.
W0521297	[Message]	Constant is too large for long long; given unsigned long long type (nonstandard).
W0521301	[Message]	*xxx* declares a non-template function -- add <> to refer to a template instance.
W0521302	[Message]	Operation may cause fixed-point overflow.
W0521307	[Message]	Class member typedef may not be redeclared.
W0521308	[Message]	Taking the address of a temporary.
W0521310	[Message]	Fixed-point value implicitly converted to floating-point type.
W0521316	[Message]	Value cannot be converted to fixed-point value exactly.
W0521319	[Message]	Fixed-point operation result is out of range.
W0521342	[Message]	const_cast to enum type is nonstandard.
W0521346	[Message]	Function returns incomplete class type %t.
W0521361	[Message]	Negation of an unsigned fixed-point value.
W0521373	[Message]	Implicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem).
W0521374	[Message]	Explicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem).

W0521375	[Message]	Conversion from pointer to same-sized integral type (potential portability problem).
W0521386	[Message]	Storage specifier ignored.
W0521396	[Message]	White space between backslash and newline in line splice ignored.
W0521400	[Message]	positional format specifier cannot be zero.
W0521420	[Message]	Some enumerator values cannot be represented by the integral type underlying the enum type.
W0521422	[Message]	Multicharacter character literal (potential portability problem).
W0521427	[Message]	offsetof applied to non-POD types is nonstandard.
W0521433	[Message]	No prior push_macro for *xxx*.
W0521443	[Message]	__real/__imag applied to real value.
W0521444	[Message]	*symbol* was declared "deprecated (*xxx*)".
W0521546	[Message]	Argument must be a constant null pointer value.
W0521547	[Message]	Insufficient number of arguments for sentinel value.
W0521548	[Message]	Sentinel argument must correspond to an ellipsis parameter.
W0521551	[Message]	No #pragma start_map_region is currently active: pragma ignored.
W0521553	[Message]	Nonstandard empty wide character literal treated as L'0'.
W0521561	[Message]	Predefined meaning of "*symbol*" discarded.
W0521564	[Message]	enum qualified name is nonstandard.
W0521565	[Message]	Anonymous union qualifier is nonstandard.
W0521566	[Message]	Anonymous union qualifier is ignored.
W0521570	[Message]	Nonstandard specifier ignored.
W0521607	[Message]	#pragma text must precede the definition of function "*function*".
	[Explanation]	#pragma text must precede the definition of function "*function*". The specification will be ignored.
W0521644	[Message]	Definition at end of file not followed by a semicolon or a declarator.
	[Explanation]	The declaration at the end of the file lacked a semicolon to indicate its termination.
W0521649	[Message]	Hite space is required between the macro name "*macro name*" and its replacement text
	[Action by User]	Insert a space between the macro name and the text to be replaced.
W0523042	[Message]	Using "*function item*" function at influence the code generation of "SuperH" compiler
	[Action by User]	The use of "*function item*" may affect compatibility with the SuperH compiler. Confirm details of differences in the specification.
W0523060	[Message]	Incompatible section specified
	[Explanation]	The same identifier was declared several times and different sections were specified for individual declarations. Only the first section declaration is valid.
W0523063	[Message]	"*character string*" has no effect in this version

W0523064	[Message]	Address taken "*variable-name*". It may cause an upset endian indirect reference.
	[Explanation]	The address of an 8-byte variable "*variable name*", which was in the different endian from that specified by the endian option, was acquired. This may lead to an indirect reference with incorrect handling of endian.
W0530809	[Message]	const qualifier for variable "*variable-name*" differs between files.
W0530811	[Message]	Type of symbol "*symbol-name*" differs between files.
W0544001	[Message]	Alignment of "*section-name*" sections is inconsistent. "*value*" is assumed.
	[Explanation]	There is an error in section naming. The same section name is specified for sections with different alignment numbers.
W0544002	[Message]	Endian of "*section-name*" sections is inconsistent. "*endian type*" is assumed.
	[Explanation]	There is an error in section naming. The same section name is specified for sections in different endian.
W0550002	[Message]	Cannot use *option1* with *option2*, option2 ignored.
	[Action by User]	Check the option specification.
W0550003	[Message]	"*option*" option needs argument, ignored.
	[Action by User]	Check the option specification parameters.
W0550004	[Message]	Illegal "*option*" option's value, ignored.
	[Action by User]	Check the option specification values.
W0550005	[Message]	Illegal "*option*" option's symbol "*symbol*", ignored.
	[Action by User]	Check the option specification symbols.
W0550006	[Message]	Illegal "*option*" option's argument, ignored.
	[Action by User]	Check the option specification parameters.
W0550007	[Message]	*option*, -C mismatch. ignore -C. output *core* common object.
	[Action by User]	Check the option specification.
W0550008	[Message]	*option* option is not supported for *core* core.
	[Explanation]	*option* option is not supported for *core* core. The option specification will be ignored.
	[Action by User]	Check the option specification.
W0550009	[Message]	Cannot find programmable peripheral I/O registers, ignored -Xprogrammable_io option.
	[Action by User]	Check the option specification.
W0550010	[Message]	Illegal displacement in *inst* instruction.
	[Explanation]	Illegal displacement in inst instruction. Only the effective lower-order digits will be recognized as being specified, and the assembly will continue.
	[Action by User]	Check the displacement value.

W0550011	[Message]	Illegal operand (range error in immediate).
	[Explanation]	Illegal operand (range error in immediate). Only the effective lower-order digits will be recognized as being specified, and the assembly will continue.
	[Action by User]	Check the immediate value.
W0550012	[Message]	Operand overflow.
	[Explanation]	Operand overflow. Only the effective lower-order digits will be recognized as being specified, and the assembly will continue.
	[Action by User]	Check the operand value.
W0550013	[Message]	*register* used as *kind* register.
	[Action by User]	Check the register specification.
W0550014	[Message]	Illegal list value, ignored.
	[Explanation]	Illegal list value, ignored. Only the effective lower-order digits will be recognized as being specified, and the assembly will continue.
	[Action by User]	Check the register list value.
W0550015	[Message]	Illegal register number, ignored.
	[Explanation]	Illegal register number, ignored. The invalid register will be ignored, and the assembly will continue.
	[Action by User]	Check the register list register.
W0550016	[Message]	Illegal operand (access width mismatch).
	[Action by User]	Check the operand's internal peripheral I/O register.
W0550017	[Message]	Base register is ep(r30) only.
	[Action by User]	Check the base register specification.
W0550018	[Message]	Illegal regID for inst.
	[Action by User]	Check the system register number.
W0550019	[Message]	Illegal operand (immediate must be multiple of 4).
	[Explanation]	Illegal operand (immediate must be multiple of 4). The number is rounded down, and assembly continues.
	[Action by User]	Check the operand value.
W0550020	[Message]	Duplicated cpu type, ignored $PROCESSOR.
	[Explanation]	The -C option is given precedence, and the target-device specified by the $PRO-CESSOR control instruction will be ignored.
	[Action by User]	Check the option specification.
W0550021	[Message]	*string* already specified, ignored.
	[Explanation]	*string* already specified, ignored. The previously specified number will be used. This specification will be ignored.
	[Action by User]	Check the number of registers.

W0550022	[Message]	Duplicated option *option*, ignored.
	[Explanation]	Duplicated option *option*, ignored. The previously specified option will be used. This specification will be ignored.
	[Action by User]	Check the option specification.
W0550023	[Message]	Start address of programmable io is out of range(0x0,*value1*-*value2*),ignored -Xprogrammable_io option.
	[Explanation]	Start address of programmable io is out of range(0x0,*value1*-*value2*),ignored -Xprogrammable_io option. The specified value will be ignored, and the initial value of the device will be used.
	[Action by User]	Check the option values.
W0550024	[Message]	Sorry, -*option option* not implemented, ignored.
	[Action by User]	Check the option specification.
W0550026	[Message]	Illegal register number, aligned odd register(r*XX*) to be even register(r*YY*).
	[Explanation]	Odd-numbered registers (r1, r3, ... r31) have been specified. The only general-purpose registered that can be specified are even-numbered (r0, r2, r4, ... r30). Assembly will continue, assuming that even-numbered registers (r0, r2, r4, ... r30) have been specified.
	[Action by User]	Check the register specification.
W0550027	[Message]	Illegal *control* value, ignored.
	[Explanation]	The *control* control instruction differs from a previous specification. The previous specification will take precedence, and register modes specified by subsequent *control* control instructions will be ignored.
	[Action by User]	Check the *control* control instruction specification.
W0550028	[Message]	Duplicated reg_mode, ignored $REG_MODE.
	[Explanation]	Duplicated reg_mode, ignored $REG_MODE. The "-Xreg_mode" option takes precedence, and register modes specified via the $REG_MODE control instruction will be ignored.
	[Action by User]	Check the option specification.
W0550029	[Message]	Can not use r0 as destination in mul/mulu in *device-name* core.
	[Action by User]	Check the operand.
W0550030	[Message]	Can not use mul/mulu X,Y,Y format in *device-name* core.
	[Action by User]	Check the operand.
W0550031	[Message]	*identifier* undefined.
	[Action by User]	Check the identifier.
W0550032	[Message]	Cache instruction is used as cll.
	[Action by User]	The use of the cache instruction as cll is not recommended. Use the cll instruction.
W0550605	[Message]	"*path-name*" specified by the "*character string*" option is a file. Specify a folder.
W0550606	[Message]	The folder "*folder-name*" specified by the "*character string*" option is not found.
W0550623	[Message]	The "*character string2*" option is ignored when the "*character string1*" option is specified at the same time.

W0550644	[Message]	"-C" option and "-Xcommon" option is mismatch. Instruction set by "*character string1*" option is ignored. Create common object for "*character string2*" instruction set.
W0550646	[Message]	"*character string1*" specified in the "*character string2* option is not allowed for a preprocessor macro. Recognized only as an assembler symbol
W0550647	[Message]	The "*character string*" option is specifiedmore than once. The latter is valid.
W0550649	[Message]	The "*character string2*" option is ignored when the "*character string1*" option and the "*character string2*" option are inconsistent.
W0550651	[Message]	The "*character string2*" option is ignored when the "*character string1*" option is not specified.
W0551000	[Message]	'.ALIGN' with not 'ALIGN' specified relocatable section.
	[Explanation]	Directive command .ALIGN is written in a section that does not have an ALIGN specification.
	[Action by User]	Check the position where directive command .ALIGN is written. Write an ALIGN specification in the section definition line of a section in which directive command .ALIGN is written.
W0551001	[Message]	Destination address may be changed.
	[Explanation]	The jump address can be a position that differs from an anticipated destination.
	[Action by User]	When writing an address in a branch instruction operand using a location symbol for offset, be sure to write the addressing mode, jump distance, and instruction format specifiers for all mnemonics at locations from that instruction to the jump address.
W0551002	[Message]	Floating point value is out of range.
	[Explanation]	The floating-point value is out of range.
	[Action by User]	Check the floating-point value written in the source code. The value out of range is ignored.
W0551003	[Message]	Location counter exceed.
	[Explanation]	The location counter value has exceeded 0FFFFFFFFh.
	[Action by User]	Check the value of the operand in .ORG. Correct the source code.
W0551004	[Message]	'.ALIGN' size is different.
	[Explanation]	The specified boundary alignment value does not match the other settings.
	[Action by User]	Check the alignment value.
W0551006	[Message]	Data in 'CODE' section align in 4byte.
	[Explanation]	When endian=big is specified, the start address of the data area in the CODE section is aligned to a 4-byte boundary.
W0551007	[Message]	Data size in 'CODE' section align in 4byte.
	[Explanation]	When endian=big is specified, the size of the data area in the CODE section is adjusted to a multiple of 4.
W0551009	[Message]	Multiple symbols.
	[Explanation]	.STACK(stack value setting) is specified multiple times for a single symbol.
W0551010	[Message]	Section attribute mismatch.
	[Explanation]	The specified section attribute does not match the other settings.

W0551011	[Message]	Use PM instruction.
	[Explanation]	A privileged instruction is used.
W0551012	[Message]	Use FPU instruction.
	[Explanation]	A floating-point operation instruction is used.
W0551013	[Message]	Use DSP instruction.
	[Explanation]	A DSP function instruction is used.
W0551014	[Message]	Too many actual macro parameters.
	[Explanation]	There are too many actual macro parameters. Extra macro parameters will be ignored.
W0551015	[Message]	Actual macro parameters are not enough.
	[Explanation]	The number of actual macro parameters is smaller than that of formal macro parameters. The formal macro parameters that do not have corresponding actual macro parameters are ignored.
W0551016	[Message]	'.END' statement is in include file.
	[Explanation]	The include file contains an .END statement. The software will ignore .END as it executes.
	[Action by User]	.END cannot be written in include files. Delete this statement.
W0551017	[Message]	Use of <instruction type> instruction found.
	[Explanation]	An instruction indicated by <instruction type> is used.
W0561000	[Message]	Option "*option*" ignored
	[Explanation]	The option named *option* is invalid, and is ignored.
W0561001	[Message]	Option "*option1*" is ineffective without option "*option2*"
	[Explanation]	*option 1* needs specifying *option 2*. *option 1* is ignored.
W0561002	[Message]	Option "*option1*" cannot be combined with option "*option2*"
	[Explanation]	*option 1* and *option 2* cannot be specified simultaneously. *option 1* is ignored.
W0561003	[Message]	Divided output file cannot be combined with option "*option*"
	[Explanation]	*option* and the option to divide the output file cannot be specified simultaneously. option is ignored. The first input file name is used as the output file name.
W0561004	[Message]	Fatal level message cannot be changed to other level : "*option*"
	[Explanation]	The level of a fatal error type message cannot be changed. The specification of *option* is ignored. Only errors at the information/warning/error level can be changed with the change_message option.
W0561005	[Message]	Subcommand file terminated with end option instead of exit option
	[Explanation]	There is no processing specification following the end option. Processing is done with the exit option assumed.
W0561006	[Message]	Options following exit option ignored
	[Explanation]	All options following the exit option is ignored.
W0561007	[Message]	Duplicate option : "*option*"
	[Explanation]	Duplicate specifications of *option* were found. Only the last specification is effective.

W0561010	[Message]	Duplicate file specified in option "*option*" : "*file name*"
	[Explanation]	*option* was used to specify the same file twice. The second specification is ignored.
W0561011	[Message]	Duplicate module specified in option "*option*" : "*module*"
	[Explanation]	*option* was used to specify the same module twice. The second specification is ignored.
W0561012	[Message]	Duplicate symbol/section specified in option "*option*" : "*name*"
	[Explanation]	*option* was used to specify the same symbol name or section name twice. The second specification is ignored.
W0561013	[Message]	Duplicate number specified in option "*option*" : "*number*"
	[Explanation]	*option* was used to specify the same error number. Only the last specification is effective.
W0561014	[Message]	License manager is not installed
W0561016	[Message]	The evaluation version of *version* is valid for the remaining *number* days. After that, link size limit (128Kbyte) will be applied. Please consider purchasing the product.
W0561017	[Message]	Paid license of "*version*" is not found, and the evaluation period has expired. Please consider purchasing the product.
W0561100	[Message]	Cannot find "*name*" specified in option "*option*"
	[Explanation]	The symbol name or section name specified in *option* cannot be found. *name* specification is ignored.
W0561101	[Message]	"*name*" in rename option conflicts between symbol and section
	[Explanation]	*name* specified by the rename option exists as both a section name and as a symbol name. Rename is performed for the symbol name only in this case.
W0561102	[Message]	Symbol "*symbol*" redefined in option "*option*"
	[Explanation]	The symbol specified by *option* has already been defined. Processing is continued without any change.
W0561103	[Message]	Invalid address value specified in option "*option*" : "*address*"
	[Explanation]	*address* specified by *option* is invalid. *address* specification is ignored.
W0561104	[Message]	Invalid section specified in option "*option*" : "*section*"
	[Explanation]	An invalid section is specified in *option*.
	[Action by User]	Confirm the following: (1) The "-output" option does not accept specification of a section that has no initial value. (2) The "-jump_entries_for_pic" option accepts specification of only a code section and no other sections.
W0561110	[Message]	Entry symbol "*sybol*" in entry option conflicts
	[Explanation]	A symbol other than *symbol* specified by the entry option is specified as the entry symbol on compiling or assembling. The option specification is given priority.
W0561120	[Message]	Section address is not assigned to "*section*"
	[Explanation]	*section* has no addresses specified for it. *section* will be located at the rearmost address.
	[Action by User]	Specify the address of the section using the rlink option "-start".

W0561121	[Message]	Address cannot be assigned to absolute section "*section*" in start option
	[Explanation]	*section* is an absolute address section. An address assigned to an absolute address section is ignored.
W0561122	[Message]	Section address in start option is incompatible with alignment : "*section*"
	[Explanation]	The address of *section* specified by the start option conflicts with memory boundary alignment requirements. The section address is modified to conform to boundary alignment.
W0561130	[Message]	Section attribute mismatch in rom option : "*section1*","*section2*"
	[Explanation]	The attributes and boundary alignment of *section 1* and *section 2* specified by the rom option are different. The larger value is effective as the boundary alignment of *section 2*.
W0561140	[Message]	Load address overflowed out of record-type in option "*option*"
	[Explanation]	A record type smaller than the address value was specified. The range exceeding the specified record type has been output as different record type.
W0561141	[Message]	Cannot fill unused area from "*address*" with the specified value
	[Explanation]	Specified data cannot be output to addresses higher than *address* because the unused area size is not a multiple of the value specified by the space option.
W0561142	[Message]	Cannot find symbol which is a pair of "*symbol*"
	[Explanation]	A "symbol" generated by the -create_unfilled_area option is not part of a pair.
W0561143	[Message]	Address range "*start address-end address*" cannot be placed in flash memory area.
	[Explanation]	The range of "*start address-end address*" is not in the flash memory area. Therefore, there is data that cannot be written by a flash programmer.
W0561150	[Message]	Sections in "*option*" option have no symbol
	[Explanation]	The section specified in *option* does not have an externally defined symbol.
W0561160	[Message]	Undefined external symbol "*symbol*"
	[Explanation]	An undefined external symbol *symbol* was referenced.
W0561181	[Message]	Fail to write "*type of output code*"
	[Explanation]	Failed to write *type of output code* to the output file. The output file may not contain the address to which *type of output code* should be output. Type of output code: When failed to write CRC code : "CRC Code"
W0561182	[Message]	Cannot generate vector table section "*section*"
	[Explanation]	The input file contains vector table *section*. The linker does not create *section* automatically.
W0561183	[Message]	Interrupt number "*vector number*" of "*section*" is defined in input file
	[Explanation]	The vector number specified by the VECTN option is defined in the input file. Processing is continued with priority given on the definition in the input file.
W0561184	[Message]	Interrupt number "*vector number*" of "*section*" is defined
W0561190	[Message]	Section "*section*" was moved other area specified in option "*cpu=<attribute>*"
	[Explanation]	The object size was modified through optimization of access to external variables. Accordingly, *section* in the area specified by the next cpu specification was moved.

W0561191	[Message]	Area of "FIX" is within the range of the area specified by "*cpu=<attribute>*" :"*<start>-<end>*"
	[Explanation]	In the cpu option, the address range of <start>-<end> specified for FIX overlapped with that specified for another memory type. The setting for FIX is valid.
W0561192	[Message]	Bss Section "*section name*" is not initialized
	[Explanation]	*section name*, which is a data section without an initial value, cannot be initialized by the initial setup program.
	[Action by User]	Check the address range specified with -cpu and the sizes of pointer variables.
W0561193	[Message]	Section "*section name*" specified in option "*option*" is ignored
	[Explanation]	*option* specified for the section newly created due to -cpu=stride is invalid.
	[Action by User]	Do not specify *option* for the newly created section.
W0561194	[Message]	Section "*section*" in relocation "*file*"-"*section*"-"*offset*" is changed.
	[Explanation]	The relocation *section file offset* now refers to a location in the new section created with the division of section.
	[Action by User]	To prevent division, declare the contiguous_section option for *section*.
W0561200	[Message]	Backed up file "*file1*" into "*file2*"
	[Explanation]	Input file *file 1* was overwritten. A backup copy of the data in the previous version of *file 1* was saved in *file 2*.
W0561300	[Message]	Option "*option*" is ineffective without debug information
	[Explanation]	There is no debugging information in the input files. The "*option*" has been ignored.
	[Action by User]	Check whether the relevant option was specified at compilation or assembly.
W0561301	[Message]	No inter-module optimization information in input files
	[Explanation]	No inter-module optimization information is present in the input files. The optimize option has been ignored.
	[Action by User]	Check whether the goptimize option was specified at compilation or assembly.
W0561302	[Message]	No stack information in input files
	[Explanation]	No stack information is present in the input files. The stack option is ignored. If all input files are assembler output files, the stack option is ignored.
W0561305	[Message]	Entry address in "*file"* conflicts : "*address*"
	[Explanation]	Multiple files with different entry addresses are input.
W0561310	[Message]	"*section*" in "*file*" is not supported in this tool
	[Explanation]	An unsupported section was present in *file*. *section* has been ignored.
W0561311	[Message]	Invalid debug information format in "*file*"
	[Explanation]	Debugging information in *file* is not dwarf2. The debugging information has been deleted.
W0561320	[Message]	Duplicate symbol "*symbol*" in "*file*"
	[Explanation]	The symbol named *symbol* is duplicated. The symbol in the first file input is given priority.

W0561321	[Message]	Entry symbol "*symbol*" in "*file*" conflicts
	[Explanation]	Multiple object files containing more than one entry symbol definition were input. Only the entry symbol in the first file input is effective.
W0561322	[Message]	Section alignment mismatch : "*section*"
	[Explanation]	Sections with the same name but different boundary alignments were input. Only the largest boundary alignment specification is effective.
W0561323	[Message]	Section attribute mismatch : "*section*"
	[Explanation]	Sections with the same name but different attributes were input. If they are an absolute section and relative section, the section is treated as an absolute section. If the read/write attributes mismatch, both are allowed.
W0561324	[Message]	Symbol size mismatch : "*symbol*" in "*file*"
	[Explanation]	Common symbols or defined symbols with different sizes were input. A defined symbol is given priority. In the case of two common symbols, the symbol in the first file input is given priority.
W0561325	[Message]	Symbol attribute mismatch : "*symbol*" : "*file*"
	[Explanation]	The attribute of *symbol* in *file* does not match the attribute of the same-name symbol in other files.
	[Action by User]	Check the symbol.
W0561326	[Message]	Reserved symbol "*symbol*" is defined in "*file*"
	[Explanation]	Reserved symbol name *symbol* is defined in *file*.
W0561327	[Message]	Section alignment in option "aligned_section" is small : "*section*"
	[Explanation]	Since the boundary alignment value specified for aligned_section is 16 which is smaller than that of *section*, the option settings made for that section are ignored.
W0561331	[Message]	Section alignment is not adjusted : "*section*"
	[Explanation]	Sections with the same name but different boundary alignment values were input. Only the largest boundary alignment specification is effective. The alignment condition at input may not be satisfied.
W0561402	[Message]	Parentheses specified in option "start" with optimization
	[Explanation]	Optimization is not available when parentheses "( )" are specified in the start option. Optimization has been disabled.
W0561410	[Message]	Cannot optimize "*file*"-"*section*" due to multi label relocation operation
	[Explanation]	A section having multiple label relocation operations cannot be optimized. Section *section* in *file* has not been optimized.
W0561430	[Message]	Cannot generate effective bls file for compiler optimization
	[Explanation]	An invalid bls file was created. This optimization is not available even if optimization of access to external variables (map option) is specified for compilation.
	[Action by User]	The optimization of access to external variables (map option) in the compiler has the following restriction. Check if this restriction is applicable and modify the section allocation. Access to external variables cannot be optimized in some cases if a data section is allocated immediately after a data section when the base option is specified for compilation. Note: The bls file indicates the external symbol allocation information file. It contains the information to be used for the map option of the compiler.

W0561510	[Message]	Input file was compiled with option "smap" and option "map" is specified at linkage
	[Explanation]	A file was compiled with smap specification.
	[Action by User]	The file with smap specification should not be compiled with the map option specification in the second build processing.
W0571600	[Message]	An error occurred during name decoding of "*instance*"
	[Explanation]	"*instance*" was not decoded. The message is output using the encoding name.
W0578306	[Message]	can not open file *file*
W0578307	[Message]	can not close file *file*
W0578308	[Message]	can not read file *file*
W0578309	[Message]	can not write file *file*
W0578310	[Message]	can not seek file *file*
W0578311	[Message]	can not find file *file*
W0578315	[Message]	can not find member *member*
	[Explanation]	Can not find member *member* in the library file.
W0578322	[Message]	this symbol offset not true
	[Explanation]	This symbol offset not true in the library file.
W0591300	[Message]	Command parameter specified twice "*option-name*"
	[Explanation]	The option that can be specified only once is specified more than once. Check the option you want to enable.
W0591301	[Message]	"*option-name*" option ignored
	[Explanation]	The specification of "*option-name*" is ignored.

10.5.6 Standard Library Error Messages

For some library functions, if an error occurs during the library function execution, an error code is set in the macro **errno** defined in the header file <**errno.h**> contained in the standard library.

Error messages are defined in the error codes so that error messages can be output. The following shows an example of an error message output program.

Example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
main()
{
    FILE *fp;

    fp=fopen("file", "w");
    fp=NULL;

    fclose(fp);                         /* error occurred */

    printf("%s\n", strerror(errno));  /* print error message */
}
```

Description:

(1) Since the file pointer of **NULL** is passed to the **fclose** function as an argument, an error will occur. In this case, an error code corresponding to **errno** is set.

(2) The **strerror** function returns a pointer of the string literal of the corresponding error message when the error code is passed as an argument. An error message is output by specifying the output of the string literal of the **printf** function.

List of Standard Library Error Messages

Error No.	Error Message/Explanation	Functions to Set Error Code
0x22 (ERANGE)	Data out of range An overflow occurred.	frexp, ldexp, modf, ceil, floor, fmod, atof, atoi, atol, atoll, atolfixed, atolaccum, strtod, strtol, strtoul, strtoll, strtoull, strtolfixed, strtolaccum, perror, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceilf, cos, cosf, cosh, coshf, exp, expf, floorf, fmodf, ldexpf, log,log10, log10f, logf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexpf
0x21 (EDOM)	Data out of domain Results for mathematical parameters are not defined.	acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceil, ceilf, cos, cosf, cosh, coshf, exp, expf, floor, floorf, fmod, fmodf, ldexp, ldexpf, log, log10, log10f, logf, modf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexp, frexpf
0x450 (ESTRN)	Too long string The length of string literal exceeds 512 characters.	atof, atoi, atol, atoll, atolfixed, atolaccum, strtod, strtol, strtoul, strtoll, strtoull, strtolfixed, strtolaccum
0x04B0 (ECBASE)	Invalid radix An invalid radix was specified.	strtol, strtoul, strtoll, strtoull
0x04B2 (ETLN)	Number too long The specified number exceeds the number of significant digits.	atof, atolfixed, atolaccum, strtod, strtolfixed, strtolaccum, fscanf, scanf, sscanf
0x04B4 (EEXP)	Exponent too large The specified exponent exceeds three digits.	atof, strtod, fscanf, scanf, sscanf
0x04B6 (EEXPN)	Normalized exponent too large The exponent exceeds three digits when the string literal is normalized to the IEEE standard decimal format.	atof, strtod, fscanf, scanf, sscanf
0x04BA (EFLOATO)	Overflow out of float A float-type decimal value is out of range (overflow).	fscanf, scanf, sscanf
0x04C4 (EFLOATU)	Underflow out of float A float-type decimal value is out of range (underflow).	fscanf, scanf, sscanf
0x04E2 (EDBLO)	Overflow out of double A double-type decimal value is out of range (overflow).	fscanf, scanf, sscanf
0x04EC (EDBLU)	Underflow out of double A double-type decimal value is out of range (underflow).	fscanf, scanf, sscanf
0x04F6 (ELDBLO)	Overflow out of long double A long double-type decimal value is out of range (overflow).	fscanf, scanf, sscanf
0x0500 (ELDBLU)	Underflow out of long double A long double-type decimal value is out of range (underflow).	fscanf, scanf, sscanf

11. Usage Notes

This chapter describes the points to be noted when using the CCRX.

11.1 Notes on Program Coding

(1) Functions with Prototype Declarations
When a function is called, the prototype of the called function must be declared. If a function is called without a prototype declaration, parameters may not be received and passed correctly.

Examples 1. The function has the **float** type parameter (when **dbl_size=8** is specified).

```
void g()
{
            float a;
            ...
            f(a);               //Converts a to double type
}
void f(float x)
{...}
```

Examples 2. The function has **signed char, (unsigned) char**, **(signed) short**, and **unsigned short** type parameters passed by stack.

```
void h();
void g()
{
            char a,b;
            ...
            h(1,2,3,4,a,b);     // Converts a and b to int type
}
void h(int a1, int a2, int a3, int a4, char a5, char a6)
{...}
```

(2) Function Declaration Containing Parameters without Type Information
When more than one function declaration (including function definition) is made for the same function, do not use both a format in which parameters and types are not specified together and a format in which parameters and types are specified together.
If both formats are used, the generated code may not process types correctly because there is a difference in how the parameters are interpreted in the caller and callee.
When the error message **C5147** is displayed at compilation, this problem may have caused it. In such a case, either use only a format in which parameters and types are specified together or check the generated code to ensure that there is no problem in parameter passing.

Example Since **old_style** is written in different formats, the meaning of the types of parameters **d** and **e** are different in the caller and callee. Thus, parameters are not passed correctly.

```
extern int old_style(int,int,int,short,short);
    /* Function declaration: Format in which parameters and types are specified
                 together */
int old_style(a,b,c,d,e)
              /* Function definition: Format in which parameters and types are not
                 specified togheer */
int a,b,c;
 short d,e;
{
    return a + b + c + d + e;
}
int result;
func()
{
    result = old_style(1,2,3,4,5);
}
```

(3) Expressions whose Evaluation Order is not Specified by the C/C++ Language
When using an expression whose evaluation order is not specified in the C/C++ language specifications, the operation is not guaranteed in a program code whose execution results differ depending on the evaluation order.

Example

```
a[i]=a[++i];  // The value on the left side differs depending on whether
              // the right side of the assignment expression is evaluated first.
sub(++i, i) ; // The value of the second parameter differs depending on whether
              // the first parameter in the function is evaluated first.
```

(4) Overflow Operation and Zero Division
Even if an overflow operation or floating-point zero division is performed, error messages will not be output. However, if an overflow operation is included in the operations of a single constant or between constants, error messages will be output at compilation.

Example

```
void main()
{
  int ia;
  int ib;
  float fa;
  float fb;

  ib=32767;
  fb=3.4e+38f;

  /* Compilation error messages are output when an overflow operation */
  /* is included in operations of a constant or between constants */

  ia=99999999999;       /* (W) Detects overflow in constant operation */
  fa=3.5e+40f;          /* (E) Detects overflow in floating-point operation */

  /* No error message is output for overflow at execution */

  ib=ib+32767;          /* Ignores overflow in operation result */
  fb=fb+3.4e+38f;       /* Ignores overflow in floating-point operation result */
}
```

(5) Writing to const Variables

Even if a variable is declared with **const** type, if assignment is done to a non-**const** type variable converted from **const** type or if a program compiled separately uses a parameter of a different type, the compiler cannot check the writing to a **const** type variable. Therefore, precautions must be taken.

Example

```
<Example>
const char *p;            /* Because the first parameter in library     */
:                         /* function strcat is a pointer to char, the  */
strcat(p, "abc");         /* area indicated by the parameter may change */

  file 1
const int i;

  file 2
extern int i;             /* In file 2, variable i is not declared as  */
:                         /* const, therefore writing to it in file 2  */
i=10;                     /* is not an error                           */
```

(6) Precision of Mathematical Function Libraries

For functions **acos(x)** and **asin(x)**, an error is large around x=1. Therefore, precautions must be taken. The error range is as follows:

Absolute error for acos(1.0 – ε) double precision 2^{-39} (ε = 2^{-33})
single precision 2^{-21} (ε = 2^{-19})

Absolute error for asin(1.0 – ε) double precision 2^{-39} (ε = 2^{-28})
single precision 2^{-21} (ε = 2^{-16})

(7) Codes that May be Deleted by Optimization

A code continuously referencing the same variable or a code containing an expression whose result is not used may be deleted as redundant codes at optimization by the compiler. Variables should be declared with **volatile** in order for accesses to always be guaranteed.

Example

```
[1] b=a;          /* The expression in the first line may be deleted */
                  /* as redundant code                               */
    b=a;
[2] while(1)a;    /* The reference to variable a and the loop   */
                  /* statement may be deleted as redundant code */
```

(8) Differences between C89 Operation and C99 Operation

In the C99, selection statements and repeat statements are enclosed in curly brackets { }. This causes operations to differ in the C89 and C99.

Example

```
<Example>
enum {a,b};
int g(void)
{
    if(!sizeof(enum{b,a}))
        return a;
    return b;
}
```

If the above code is compiled with **-lang=c99** specified, it is interpreted as follows:

```
enum {a,b};
int g(void)
{
     {
         if(!sizeof(enum{b,a}))
             return a;
     }
     return b;
}
```

g()=0 in **-lang=c** becomes **g()=1** in **-lang=c99**.

(9) Operations and Type Conversions That Lead to Overflows
The result of any operation or type conversion must be within the allowed range of values for the given type (i.e. values must not overflow). If an overflow does occur, the result of the operation or type conversion may be affected by other conditions such as compiler options.
In the standard C language, the result of an operation that leads to an overflow is undefined and thus may differ according to the current conditions of compilation. Ensure that no operations in a program will lead to an overflow.
The following example illustrates this problem.

Example Type conversion from **float** to **unsigned short**

```
float f = 2147483648.0f;
unsigned short ui2;
void ex1func(void)
{
       ui2 = f;  /* Type conversion from float to unsigned short */
}
```

The value of **ui2**, which is acquired as the result of executing **ex1func**, depends on whether **–fpu** or **–nofpu** has been specified.
-fpu (with the FPU): ui2 = 65535
-nofpu (without the FPU): ui2 = 0
This is because the method of type conversion from **float** to **unsigned short** differs according to whether **–fpu** or **–nofpu** has been specified.

(10) Symbols That Contain Two or More Underscores (__)
Symbols must not contain sequences of two or more underscores. Even though the code generated in such cases seems normal, the symbol names may be mistaken as different C++ function names when they are output as linkage-map information.

Example

```
int sample__Fc(void) { return 0; }
```

This will be output to the linkage map as **sample(char)** rather than _**sample__Fc**.

11.2 Notes on Compiling a C Program with the C++ Compiler

(1) Functions with Prototype Declarations
Before using a function, a prototype declaration is necessary. At this time, the types of the parameters should also be declared.

```
extern void func1();
void g()
{
    func1(1); // Error
}
```

```
extern void func1(int);
void g()
{
    func1(1); // OK
}
```

(2) Linkage of const Objects
Whereas in C programs **const** type objects are linked externally, in C++ programs they are linked internally. In addition, **const** type objects require initial values.

```
const cvalue1;     // Error

const cvalue2 = 1; // Links internally
```

```
const cvalue1 = 0;        // Gives initial value

extern const cvalue2 = 1; // Links externally as a C program
```

(3) Assignment of void*
In C++ programs, if explicit casting is not used, assignment of pointers to other objects (excluding pointers to functions and to members) is not possible.

```
void func(void *ptrv, int *ptri)
{
    ptri = ptrv; // Error
}
```

```
void func(void *ptrv, int *ptri)
{
    ptri = (int *)ptrv; // OK
}
```

11.3 Notes on Options

(1) Options Requiring the Same Specifications
Options that should always be specified in the same way are shown in (a) and (b) below. If relocatable files and library files using different options are linked, the operation of the program at runtime is not guaranteed.

(a) The five options **isa**, **cpu**, **endian**, **base**, and **fint_register** should be specified in the same way in the compiler, assembler, and library generator.

(b) The options in the Microcontroller Options section of the COMMAND REFERENCE chapter, except for the options in (a), must be specified in the same way in the compiler and library generator.

(2) When Using -reent (an Option Which Generates a Reentrant Library) of lbgrx (a Library Generator)
To enable specification of the -reent option for the lbgrx library generator in a project generated in a Renesas integrated development environment, confirm if the low-level _INIT_IOLIB() function of the project contains the statements for execution in relation to alignment_Files listed below.
If the function does not include these statements, add them with reference to the contents of the _INIT_IOLIB() function in the listing of the lowsrc.c file in the of section 8.4 Coding Example.
_Files[0] = stdin;
_Files[1] = stdout;
_Files[2] = stderr;

11.4 Preventing E0562330 Errors in Cases Where Optimization by the Optimizing Linkage Editor is Enabled

The optimizing linkage editor is used for optimization, which involves replacing existing code with code that takes up less space in relation to the address where each symbol referred to by the original instruction is located. To reduce the size of the code as much as possible, the linkage editor will attempt to replace numerous instructions.

Replacing instructions may, however, change the addresses where individual symbols are located and thus prevent reference to symbols by the optimized code. In such cases, the optimizing linkage editor reports an E0562330 error and stops operations so that code having this problem will not be output.

Due to the characteristics of the RX architecture, this error may arise when any symbol (e.g. a variable, constant, or switch table) located at FFFF8000h or a higher address prior to optimization is allocated to an address below FFFF8000h after optimization.

(1) Outline

Examples of conditions leading to E0562330 errors are given below.

In a program where the reading of constants CONST1 and CONST2 is intended, section P is followed by section C, which is allocated to address FFFF8000h prior to optimization.

Examples 1.

Section P

```
; 6-byte instruction MOV.L #_CONST1:32, R1
MOV.L #0FFFF8002H, R1

; 6-byte instruction MOV.L #_CONST2:32, R2
MOV.L #0FFFF8006H, R2
```

Section C

```
_CONST0:                    ; Allocated to address FFFF8000h
    .byte 00H,01H
_CONST1:                    ; Allocated to address FFFF8002h
    .byte "123"
    .byte 00H
_CONST2:                    ; Allocated to address FFFF8006h
    .byte "abc"
    .byte 00H
```

Since optimization changes the address of section C from FFFF8000H (a 32-bit immediate value) to -8000H (a signed 16-bit immediate value), the 32-bit immediate-value transfer MOV.L instruction will be replaced by a 16-bit immediate-value transfer instruction.
This reduces the size of section P and leads to its allocation to a lower address than section C.

Examples 2.

Section P

```
; 4-byte instruction MOV.L #_CONST1:16, R1
MOV.L #-8002H, R1 ; Exceeds the range of signed 16-bit values

; 4-byte instruction MOV.L #_CONST2:16, R2
MOV.L #-7FFEH, R2
```

Section C

```
_CONST0:                    ; Allocated to address FFFF7FFCh
    .byte 00H,01H
_CONST1:                    ; Allocated to address FFFF7FFEh
    .byte "123"
    .byte 00H
_CONST2:                    ; Allocated to address FFFF8002h
    .byte "abc"
    .byte 00H
```

Due to optimization of multiple MOV.L instructions, constant CONST1 is allocated to an address beyond the range of signed 16-bit values, leading to an E0562330 error as shown below.

Examples 3.

```
E0562330:Relocation size overflow : "fileA.obj"-"P"-"0000002"
```

(2) How to identify the cause

You can identify the cause of the error in the following way.

First, specify the optimizing linkage editor options **nooptimize** and **list** and build the user program to generate a link map file. Since the information on the allocation of sections begins from the line "*** Mapping List ***" in the link map file, check sections allocated to FFFF8000h and higher addresses.

Examples 4.

```
*** Mapping List ***
C
                                ffff7ffc  ffff8005        a    1
```

After that, specify the assembler option **listfile** and build the user program to generate a source listing file. Find the section (P in example 3) in the source listing file for the object file ("fileA.obj" in example 3) indicated by the error message.

The offset value indicated by the error message is the value after optimization.

In the source listing file, locate the instructions that are at or above the offset address (0000002 in example 3) and references to any symbols within sections allocated to the address FFFF8000h or a higher address.

In example 5, you can see that the MOV.L instruction that attempts access to constant CONST1 in section C at an address above FFFF8000h is the reason for the error.

Examples 5.

```
                .section P, CODE
00000000 FB1Arrrr            MOV.L #_CONST1:16, R1 <- Access to section C
00000004 FB2Arrrr            MOV.L #_CONST2:16, R2

                .section C, ROMDATA
00000000             _CONST0:
00000000 0001           .byte 00H,01H
00000002             _CONST1:
00000002 313233         .byte "123"
00000005 00             .byte 00H
00000006             _CONST2:
00000006 616263         .byte "abc"
00000009 00             .byte 00H
```

(3) Countermeasures

Select one of the following countermeasures.

- Countermeasure 1: Change the order of sections

  If the first address of a section allocated to address FFFF8000h or above before optimization is shifted to an address below FFFF8000h after optimization, use the start option to change the place of that section in the order of sections.

  Example

  Before:

  ```
  -start=P,C,L,D/FFFF7000
  ```

  After (switching sections C and L):

  ```
  -start=P,L,C,D/FFFF7000
  ```

  If an E0562330 error is still output even after the order of sections is switched, change the order again until the error is no longer output.

- Countermeasure 2: Allocate the section to address FFFF8000h
  If the first address of a section allocated to address FFFF8000h or above before optimization is shifted to an address below FFFF8000h after optimization, use the **start** option to allocate that section to address FFFF8000h.

  Example
  Before:
  ```
  -start=P,C,L,D/FFFF7000
  ```
  After:
  ```
  -start=P/FFFF7000,C,L,D/FFFF8000
  ```

- Countermeasure 3: Disable optimization by the optimizing linkage editor
  Use the -**nooptimize** option to completely disable optimization or select the desired sub-options for the -**optimize** option to prevent the output of the E0562330 error while watching the reduction of the size of code by optimization.

11.5 Compatibility with an Older Version or Older Revision

The effect of the compatibility regarding a version change or revision change is described here.

11.5.1 V.1.01 and Later Versions (Compatibility with V.1.00)

(1) Changing Specifications of Intrinsic Functions
For intrinsic functions having parameters or return values that indicate addresses, their type is changed from the conventional **unsigned long** to **void ***. The changed functions are shown in Table 11.1.

Table 11.1 List of Intrinsic Functions Whose Type is Changed

No.	Item	Specification	Function	Changed Contents	
				Item	Details
1	User stack pointer (USP)	void set_usp(void *data)	USP setting	Parameter	unsigned long → void *
2		void *get_usp(void)	USP reference	Return value	unsigned long → void *
3	Interrupt stack pointer (ISP)	void set_isp(void *data)	ISP setting	Parameter	unsigned long → void *
4		void *get_isp(void)	ISP reference	Return value	unsigned long → void *
5	Interrupt table register (INTB)	void set_intb (void *data)	INTB setting	Parameter	unsigned long → void *
6		void *get_intb(void)	INTB reference	Return value	unsigned long → void *
7	Backup PC (BPC)	void set_bpc(void *data)	BPC setting	Parameter	unsigned long → void *
8		void *get_bpc(void)	BPC reference	Return value	unsigned long → void *
9	Fast interrupt vector register (FINTV)	void set_fintv(void *data)	FINTV setting	Parameter	unsigned long → void *
10		void *get_fintv(void)	FINTV reference	Return value	unsigned long → void *

Due to this change, a program using the above functions in V.1.00 may generate a warning or an error about invalid types. In this case, add or delete the cast to correct the types.
An example of a startup program normally used in V.1.00 is shown below. This example will output warning message W0520167 in V.1.01, but this warning can be avoided by deleting the cast to correct the type.

Example
[Usage example of **set_intb** function]

```
#include <machine.h>
#pragma entry Reset_Program
void PowerON_Reset_PC(void)
{
     ...
     set_intb((unsigned long)__sectop("C$VECT")); //Warning W0520167 is output
     ...
}
```

[Example of code changed to match V.1.01]

```
#include <machine.h>
#pragma entry Reset_Program
void PowerON_Reset_PC(void)
{
        ...
        set_intb(__sectop("C$VECT")); //Cast (unsigned long) is deleted
        ...
}
```

(2) Adding Section L (section Option and Start Option)
V.1.01 is provided with section **L** which is used for storing literal areas, such as, string literal.
Since the number of sections has increased and section **L** is located at the end at linkage, the optimizing linkage editor may output address error F0563100 in some cases.
To avoid such an error, adopt either one of the following methods.

(a) Add **L** to the section sequence specified with the **Start** option of the optimizing linkage editor at linkage.

Example
[Example of specification in V.1.00]

```
-start=B_1,R_1,B_2,R_2,B,R,SU,SI/01000,PResetPRG/0FFFF8000,C_1,C_2,C,C$*,D*,P,
PIntPRG,W*/0FFFF8100,FIXEDVECT/0FFFFFFD0
```

[Changed example (**L** is added after **C**)]

```
-start=B_1,R_1,B_2,R_2,B,R,SU,SI/01000,PResetPRG/0FFFF8000,C_1,C_2,C,L,C$*,D*,
P,PIntPRG,W*/0FFFF8100,FIXEDVECT/0FFFFFFD0
```

(b) Select **-section=L=C** at compilation.
By specifying **-section=L=C** at compilation, the output destination of the literal area is changed to section **C**, and a section configuration compatible with V.1.00 can be achieved.
Note that this method may affect code efficiency compared to the above method of changing the **Start** option at linkage.

11.5.2 V.2.00 and Later Versions (Compatibility with Versions between 1.00 and 1.02)

(1) Restriction That Applies to Operation of the Linkage Editor When the -merge_files Option of the Compiler Has been Used
When an object module file created by the compiler with the **-merge_files** option specified is to be linked, correct operation is not guaranteed if the **-delete**, **-rename**, or **-replace** option is specified.

(2) Note on Generation of Code That Corresponds to if Statements When optimize=0
In this version of this compiler, if statements where the conditional expression has a constant value and statements that will accordingly never be executed are not reflected in the output code whether or not optimize=0 is specified.
In the examples below, lines marked [Deleted] are not reflected at the time of code generation.

Examples 1. Expression that produces a constant value

```
int a,b,c;
void func01(void)
{
    if (1+2) {   /* [Deleted] */
        /* Executed */
        a = b;
    } else {
        /* Never executed */
        a++;     /* [Deleted] */
        b = c;   /* [Deleted] */
    }
}
```

Examples 2. Constant expressions that include symbolic addresses are also treated as constant expressions.

```
void f1(void),f2(void);
void func02(void)
{
    if (f1==0) {  /* [Deleted] */
        /* Never executed */
        f2();     /* [Deleted] */
    } else {
        /* Executed */
        f1();
    }
}
```

(3) Differences in Assembly Source Code Output by -show=source
There are the following differences in the assembly source code to be output by this version of this compiler when -**show=source** is specified.

- **.LINE** is not displayed unless -debug has been specified.
- **#include** statements are not expanded.
- The instruction that corresponds to source code that follows **#line** may be incorrect.

11.5.3 V.2.03 and Later Versions (Compatibility with Versions between 1.00 and 2.02)

(1) Const-type Static Variables without Initial Values
In versions before 2.02, const-type static variables with initial values were output first, but from this revision, const-type static variables are aligned in the data area in order of their definition regardless of the existence of initial values.

Example

```
const int a=1;
const int b;
const int c=2;
```

[Result of compilation for versions before 2.02.00]

```
    .SECTION C,ROMDATA,ALIGN=4
_a:
    .lword 00000001H
_c:
    .lword 00000002H ; The variables with initial values are output first.
_b:
    .lword 00000000H
```

[Result of compilation for versions after 2.03.00]

```
    .SECTION C,ROMDATA,ALIGN=4
_a:
    .lword 00000001H
_b:
    .lword 00000000H ;  The variables are output in order of their definition
                     ;  regardless of the existence of initial values.
_c:
    .lword 00000002H
```

11.5.4 V2.06 and Later Versions (Compatibility with V2.05 and earlier)

(1) Introducing the Method for Controlling the Output of Bit Manipulation Instructions that Involve Memory Access
In V2.05 and earlier versions of the compiler, except for intrinsic functions, there was no method for the user to always output bit manipulation instructions that involve memory access.
In V2.06, the compiler is modified to allow the user to control whether to output bit manipulation instructions that involve memory access, without using intrinsic functions.

To let the compiler output bit manipulation instructions that involve memory access without using intrinsic functions, create a source program that satisfies all the following conditions.
(a) A constant value is assigned.
(b) The value is assigned to a single-bit bit field of a 1-byte type.
(c) The bit field where the value is assigned is qualified with **volatile**.

To suppress the output of bit manipulation instructions that involve memory access, satisfy condition (c) above. Then, either assign a value that is not a constant in condition (a) or use a type that is not a 1-byte type in condition (b).

Otherwise, whether to output bit manipulation instructions that involve memory access is determined by the compiler according to the specified optimization level and the contents of the source program.

Note 1-byte types are **char**, **unsigned char**, **signed char**, **_Bool**, and **bool**. **_Bool** and **bool** are excluded when **-lang=c** is specified.

Example

```
volatile struct {
  unsigned char bit0:1;
  unsigned int  bit1:1;
} data;

void func(void) {
  data.bit0 = 1;  /* A bit manipulation instruction involving memory access is out-
put. */
  data.bit1 = 1;  /* A bit manipulation instruction involving memory access is not
output. */
}
```

To always output bit manipulation instructions that involve memory access in V2.05, use intrinsic functions **__bclr**(), **__bset**(), and **__bnot**(). When the intrinsic functions are not used, whether to output bit manipulation instructions that involve memory access is determined by the compiler according to the specified optimization level and the contents of the source program.
The V2.04 and earlier versions of the compiler do not support these intrinsic functions. To check whether bit manipulation instructions that involve memory access are output, refer to the assembly source code output by the compiler.

11.5.5 Version of Compiler Package

When using an optimizing linker, use one provided with the same compiler package used to generate all object files, relocatable files, and library files that are to be input. An optimizing linker provided with a newer compiler package can also be used.

When using standard library functions, use those provided with the same compiler package as the optimizing linker in use.

11.6 W0523041 message [C/C++ compiler]

When the -int_to_short option is specified and a file including a C standard header is compiled as C++ or EC++, the compiler may show the W0523041 message. In this case, simply ignore the message because there are no problems.

[NOTE]

In compilation of C++ or EC++, the -int_to_short option will be invalid.

Data that are shared between C and C++ (EC++) program must be declared as the long or short type rather than as the int type.

11.7 Using MVTC or POPC instructions [Assembler]

In the assembly language, the program counter (PC) cannot be specified for MVTC or POPC instructions.

11.8 Using the -delete option for linkage [Optimizing linkage editor]

When a function symbol is removed by the -delete option, its following function in the source program is not allowed to have a breakpoint at its function name on the editor while debugging. If you intend to set a breakpoint via the [Label] window at the function entrance, set the breakpoint via the [Label] window or at the program code of the function.

11.9 Path names

Absolute paths that include drive letters or relative paths can be used as the path names for specifying input/output files or folders. Each path name should consist of no more than 259 characters.

A. QUICK GUIDE

This chapter describes programming methods and the usage of extended functions for effective use of the RX family.

A.1 Variables (C Language)

This section describes variables (C language).

A.1.1 Changing Mapped Areas

The defaults for the mapped sections of variables are as follows:

- Variables without initial values: Sections **B**, **B_2**, and **B_1**
- Variables with initial values: Sections **D**, **D_2**, and **D_1** (ROM) and sections **R**, **R_2**, and **R_1** (RAM)
- **const** variables: Sections **C**, **C_2**, and **C_1**

For changing the area (section) to map variables, specify the section type and section name through **#pragma section**.

```
#pragma section <section type> <section name>
```

Variable declaration/definition

```
#pragma section
```

When a section type is specified, only section names of the specified type can be changed.
Note that in the RX family C/C++ compiler, the section to map a variable depends on the alignment value of the variable.

Example
B: Variables without initial values and an alignment value of four bytes are mapped
B_2: Variables without initial values and an alignment value of two bytes are mapped
B_1: Variables without initial values and an alignment value of one byte are mapped

For variables with initial values, the initial value is mapped to ROM and the variable itself is mapped to RAM (both ROM and RAM areas are necessary). When the **resetprg.c** file of the startup routine is used, the **INITSCT** function copies the initial values in ROM to the variables in RAM.

The relationship between the section type and the created section is shown in the following.

Name	Section Name	Attribute	Format Type	Initial Value and Write Operation	Alignment Value
Constant area	C*1*2	romdata	Relative	Has initial values and writing is not possible	4 bytes
	C_2*1*2	romdata	Relative	Has initial values and writing is not possible	2 bytes
	C_1*1*2	romdata	Relative	Has initial values and writing is not possible	1 byte
Initialized data area	D*1*2	romdata	Relative	Has initial values and writing is possible	4 bytes
	D_2*1*2	romdata	Relative	Has initial values and writing is possible	2 bytes
	D_1*1*2	romdata	Relative	Has initial values and writing is possible	1 byte

Name	Section Name	Attribute	Format Type	Initial Value and Write Operation	Alignment Value
Uninitialized data area	B[*1*2]	data	Relative	Does not have initial values and writing is possible	4 bytes
	B_2[*1*2]	data	Relative	Does not have initial values and writing is possible	2 bytes
	B_1[*1*2]	data	Relative	Does not have initial values and writing is possible	1 byte
switch statement branch table area	W[*1*2]	romdata	Relative	Has initial values and writing is not possible	4 bytes
	W_2[*1*2]	romdata	Relative	Has initial values and writing is not possible	2 bytes
	W_1[*1*2]	romdata	Relative	Has initial values and writing is not possible	1 byte
C++ initial processing/ postprocessing data area	C$INT	romdata	Relative	Has initial values and writing is not possible	4 bytes
C++ virtual function table area	C$VTBL	romdata	Relative	Has initial values and writing is not possible	4 bytes
Absolute address variable area	$ADDR_ <section>_ <address>[*3]	data	Absolute	Has or does not have initial values and writing is possible or not possible[*4]	—
Variable vector area	C$VECT	romdata	Relative	Does not have initial values and writing is possible	—

Example 1. Section names can be switched by the **section** option or the **#pragma section** extension. However, partial data (e.g., string literal) is not affected by **#pragma section**. For details, see the detailed description of 4.2.3 #pragma Directive **- #pragma section**.

Example 2. Specifying a section with an alignment value of 4 when switching the section names also changes the section name of sections with an alignment value of 1 or 2. When **#pragma endian** is used to specify an endian that differs from the setting by the **endian** option, a dedicated section is created and the relevant data stored. For this section, after the section name, **_B** is added for **#pragma endian big** and **_L** is added for **#pragma endian little**. However, partial data (e.g., string literal) is not affected by **#pragma endian**. For details, see the detailed description of 4.2.3 #pragma Directive **- #pragma endian**.

Example 3. **<section>** is a **C**, **D**, or **B** section name, and **<address>** is an absolute address (hexadecimal).

Example 4. The initial value and write operation depend on the attribute of **<section>**.

A.1.2 Defining Variables Used at Normal Processing and Interrupt Processing

Variables used for both normal processing and interrupt processing must be **volatile** qualified.

When a variable is qualified with the **volatile** qualifier, that variable is not to be optimized and optimization, such as assigning it to a register, is not performed. When operating a variable that has been **volatile** qualified, a code that reads its value from memory and writes its value to memory after operation must be used. A variable not **volatile** qualified is assigned to a register by optimization, and the code that loads that variable from memory may be deleted. When the same value is to be assigned to a variable that is not **volatile** qualified, the processing may be interpreted as redundant and the code deleted by optimization.

A.1.3 Generating a Code that Accesses Variables in the Declared Size

When accessing a variable in its declared size, the **__evenaccess** extended function should be used.

The __**evenaccess** declaration guarantees access in the size of the variable type. The guaranteed size is a scalara type (**signed char**, **unsigned char**, **signed short**, **unsigned short**, **signed int**, **unsigned int**, **signed long**, or **unsigned long**) of four bytes or less.

The __**evenaccess** is invalid to the case of accessing of members by a lump of these structure and union frame.

When a structure or union is specified, the __**evenaccess** declaration is effective for all members. In such a case, the access size of a scalara type member of four bytes or less is guaranteed but the access size for the whole structure or union is not guaranteed.

[Example]

C source code

```
#pragma address A=0xff0178
unsigned long __evenaccess A;
void test(void)
{
     A &= ~0x20;
}
```

Output code (when __**evenaccess** is not specified)

```
_test:
     MOV.L #16712056,R1
     BCLR #5,[R1] ; 1-byte memory access
     RTS
```

Output code (when __**evenaccess** is specified)

```
_test:
     MOV.L #16712056,R1
     MOV.L [R1],R5 ; 4-byte memory access
     BCLR #5,R5
     MOV.L R5,[R1] ; 4-byte memory access
     RTS
```

A.1.4 Performing const Declaration for Variables with Unchangeable Initialized Data

A variable with an initial value is normally transferred from a ROM area to a RAM area at startup, and processing is performed using the RAM area. Accordingly, if the value is initialized data which is unchangeable in the program, the allocated RAM area goes to waste. If the **const** operator is added to initialized data, transfer to the RAM area at startup is disabled and the amount of used memory can be saved.

In addition, writing a program based on the rule of not changing the initial values facilitates creation of ROM images.

[Example before improvement]

char a[] = { 1, 2, 3, 4, 5 };

Initial values are transferred from ROM to RAM and then processing is performed.

[Example after improvement]

const char a[] = { 1, 2, 3, 4, 5 };

Processing is performed using the initial values in ROM.

A.1.5 Defining the const Constant Pointer

The pointer is interpreted differently according to where "**const**" is specified.

[Example 1]

const char *p;

In this example, the object (***p**) indicated by the pointer cannot be changed. The pointer itself (**p**) can be changed. Therefore, the result becomes as shown below and the pointer itself is mapped to RAM (section **B**).

p = 0; / Error */

p = 0; /* Correct */

[Example 2]

char *const p;

In this example, the pointer itself (**p**) cannot be changed. The object (***p**) indicated by the pointer can be changed. Therefore, the result becomes as shown below and the pointer itself is mapped to ROM (section **C**).

```
*p = 0; /* Correct */
 p = 0; /* Error */
```

[Example 3]

```
char *const p;
```

In this example, the pointer itself (**p**) and the object (***p**) indicated by the pointer cannot be changed. Therefore, the result becomes as shown below and the pointer itself is mapped to ROM (section **C**).

```
*p = 0; /* Error */
 p = 0; /* Error */
```

A.1.6 Referencing Addresses of a Section

The addresses and size of a section can be referenced by using section address operators.

__sectop ("<section name>"): References the start address of <section name>

__secend ("<section name>"): References the sum of the size of <section name> and the address where <section name> starts.

__secsize ("<section name>"): References the size of <section name>

[Example]

```
#pragma section $DSEC
static const struct {
   void *rom_s; /* Acquires the start address value of the initialized data section in
ROM */
   void *rom_e; /* Acquires the last address value of the initialized data section in
ROM */
   void *ram_s; /* Acquires the start address value of the initialized data section in
RAM */
} DTBL[]={__sectop("D"), __secend("D"), __sectop("R")};
```

The **INITSCT** function in the **resetprg.c** file of the startup routine executes transfer from ROM to RAM and initialization of uninitialized areas. The addresses acquired by **__sectop** and **__secend** written in the **dbsct.c** file are referenced during execution.

A.2 Functions

This section describes functions.

A.2.1 Filling Assembler Instructions

In the RX family C/C++ compiler, assembler instructions can be written in a C-language source program using **#pragma inline_asm**.

[Example]

```
#pragma inline_asm func
static int func(int a, int b){
     ADD R2,R1 ; Assembly-language description
}
main(int *p){
     *p = func(10,20);
}
```

Inline expansion is performed for an assembly-language function specified by **#pragma inline_asm**.
The general function calling rules are also applied to the calls of assembly-language inline functions.

A.2.2 Performing In-Line Expansion of Functions

#pragma inline declares a function for which inline expansion is performed.

The compiler options **inline** and **noline** are also used to enable or disable inline expansion. However, even when the **noinline** option is specified, inline expansion is done for the function specified by **#pragma inline**.

A global function or a static function member can be specified as a function name. When inline expansion is performed for a function specified by **#pragma inline** or a function with the **inline** function specifier (C++ and C (C99), the body of the function is expanded where the function is called.

[Example]

C source code

```
#pragma inline(func)
static int func (int a, int b)
{
     return (a+b)/2;
}
int x;
main()
{
     x=func(10,20);
}
```

Expanded image

```
int x;
main()
{
     int func_result;
     {
          int a_1=10, b_1=20;
          func_result=(a_1+b_1)/2;
     }
     x=func_result;
}
```

A.2.3 Performing (Inter-File) In-Line Expansion of Functions

Normally, inline expansion is performed for functions within a file. However, using the **-file_inline** option of the compiler allows inline expansion to be performed for even inter-file function calling.

[Example]

```
<a.c>
 func(){
     g();
 }
<b.c>
 g(){
     h();
}
```

By compiling with the specification of ccrx -inline -file_inline=b.c a.c, calling of function g in a.c is expanded and becomes as follows:

```
func(){
     h();
}
```

A.3 Using Microcomputer Functions

This section describes usage of microcomputer functions.

A.3.1 Processing an Interrupt in C Language

Use **#pragma interrupt** to declare an interrupt function.

[Example]

C source code

```
#pragma interrupt func
void func(){ .... }
```

Generated code

```
_func:
     PUSHM R1-R3 ; Saves registers used in the function
          ...
     (R1, R2, and R3 are used in the function)
          ...
     POPM R1-R3 ; Restores registers that were saved at the function entry
     RTE
```

A.3.2 Using CPU Instructions in C Language

The compiler provides the following intrinsic functions for cases of accessing control registers and special instructions that cannot be expressed in C language. Refer to "4.2.6 Intrinsic Functions".

A.4 Variables (Assembly Language)

This section describes variables (assembly language).

A.4.1 Defining Variables without Initial Values

Allocate a memory area in a **DATA** section.

To define a **DATA** section, use the **.SECTION** directive. To allocate a memory area, use the **.BLKB** directive for specification in 1-byte units, the **.BLKW** directive for 2-byte units, the **.BLKL** directive for 4-byte units, and the **.BLKD** directive for 8-byte units.

[Example]

```
        .SECTION area,DATA
work1:  .BLKB 1; Allocates a RAM area in 1-byte units
work2:  .BLKW 1; Allocates a RAM area in 2-byte units
work3:  .BLKL 1; Allocates a RAM area in 4-byte units
work4:  .BLKD 1; Allocates a RAM area in 8-byte units
```

A.4.2 Defining a cost Constant with an Initial Value

Initialize a memory area in a **ROMDATA** section.

To define a **ROMDATA** section, use the **.SECTION** directive. To initialize memory, use the **.BYTE** directive for 1 byte, the **.WORD** directive for 2 bytes, the **.LWORD** directive for 4 bytes, the **.FLOAT** directive for floating-point 4 bytes, and the **.DOUBLE** directive for floating-point 8 bytes.

[Example]

```
        .SECTION value,ROMDATA
work1:  .BYTE "data";  Stores 1-byte fixed data in ROM
work2:  .WORD "data";  Stores 2-byte fixed data in ROM
work3:  .LWORD "data"; Stores 4-byte fixed data in ROM
work4:  .FLOAT 5E2;    Stores 4-byte floating-point data in ROM
work5:  .DOUBLE 5E2;   Stores 8-byte floating-point data in ROM
```

A.4.3 Referencing the Address of a Section

The size and start address of a section that were specified as operands using the **SIZEOF** and **TOPOF** operators are handled as values.

[Example]

```
...
MVTC        #(TOPOF SU + SIZEOF SU),USP
; Sets the user stack area address to USP as (SU start address + SU size)
MVTC        #(TOPOF SI + SIZEOF SI),ISP
; Sets the interrupt stack area address to ISP as (SI start address + SI size)
 ...
```

A.5 Startup Routine

This section describes the startup routine.

A.5.1 Allocating Stack Areas

Since the **PowerON_Reset_PC** function in the **resetprg.c** file of the startup routine is declared by "**#pragma entry**", the compiler and optimizing linkage editor automatically generate the initialization code for the user stack **USP** and interrupt stack **ISP** at the top of the function, based on the settings below.

(1) Setting the User Stack
Specify the size of the stack area by **#pragma stacksize su=0xXXX** in the **stacksct.h** file, and specify the location of the **SU** section by the **-start** option of the optimizing linkage editor.

(2) Setting the Interrupt Stack
Specify the size of the stack area by **#pragma stacksize si=0xXXX** in the **stacksct.h** file, and specify the location of the **SI** section by the **-start** option of the optimizing linkage editor.
[Example]

```
<resetprg.c>
...
#pragma section ResetPRG
#pragma entry PowerON_Reset_PC
void PowerON_Reset_PC(void)
{
...
<stacksct.h>
#pragma stacksize su=0x300
#pragma stacksize si=0x100
```

[Generated code example]

```
When // -start=SU,SI/01000 is specified
_PowerON_Reset_PC         MVTC        #00001300H,USP
                          MVTC        #00001400H,ISP
...
```

A.5.2 Initializing RAM

The **_INITSCT** function in the **resetprg.c** file of the startup routine is used to initialize uninitialized areas. To add a section to be initialized, add the following description to the **dbsct.c** file.
[Example]

```
<dbsct.c>
...
#pragma section C C$BSEC
extern const struct {
     _UBYTE *b_s;         /* Start address of non-initialized data section */
     _UBYTE *b_e;         /* End address of non-initialized data section */
}   _BTBL[] = {
     { __sectop("B"), __secend("B") },
     { __sectop("B_2"), __secend("B_2") },
     { __sectop("B_1"), __secend("B_1") }
};
...
```

In the above example, the addresses used in the **INITSCT** function are stored in the table in order to initialize the **B**, **B_2**, and **B_1** sections.

A.5.3 Transferring Variables with Initial Values from ROM to RAM

The **_INITSCT** function in the **resetprg.c** file of the startup routine is used to transfer variables with initial values from ROM to RAM. To add a section to be transferred, add the following description to the **dbsct.c** file.

[Example]

```
<dbsct.c>
...
#pragma section C C$DSEC
extern const struct {
     _UBYTE *rom_s;       /* Start address of the initialized data section in ROM */
     _UBYTE *rom_e;       /* End address of the initialized data section in ROM */
     _UBYTE *ram_s;       /* Start address of the initialized data section in RAM */
}   _DTBL[] = {
     { __sectop("D"), __secend("D"), __sectop("R") },
     { __sectop("D_2"), __secend("D_2"), __sectop("R_2") },
     { __sectop("D_1"), __secend("D_1"), __sectop("R_1") }
};
...
```

In the above example, the addresses used in the **INITSCT** function are stored in the table in order to transfer the contents of the **D**, **D_2**, and **D_1** sections to the **R**, **R_2**, and **R_1** sections. Note that the location addresses of the **D**, **D_2**, **D_1**, **R**, **R_2**, and **R_1** sections should be specified by the
-start option of the optimizing linkage editor. The relocation solution by transferring data from ROM to RAM should be specified by the **-rom** option of the optimizing linkage editor.

A.6 Reducing the Code Size

This section describes code size reduction.

A.6.1 Data Structure

In a case where related data is referenced many times in the same function, usage of a structure will facilitate generation of a code using relative access, and an improvement in efficiency can be expected. The efficiency will also be improved when data is passed as arguments. Because the access range of relative access is limited, it is effective to place the frequently accessed data at the top of the structure.

When data takes the form of a structure, it is easy to perform tuning that changes the data expressions.

[Example]

Numeric values are assigned to variables **a**, **b**, and **c**.

Source code before improvement

```
int a, b, c;
void func()
{
     a = 1;
     b = 2;
     c = 3;
}
```

Assembly-language expansion code before improvement

```
_func:
     MOV.L #_a,R4
     MOV.L #00000001H,[R4]
     MOV.L #_b,R4
     MOV.L #00000002H,[R4]
     MOV.L #_c,R4
     MOV.L #00000003H,[R4]
     RTS
```

Source code after improvement

```
struct s{
     int a;
     int b;
     int c;
} s1;
void func()
{
     register struct s *p=&s1;
     p->a = 1;
     p->b = 2;
     p->c = 3;
}
```

Assembly-language expansion code after improvement

```
_func:
     MOV.L #_s1,R5
     MOV.L #00000001H,[R5]
     MOV.L #00000002H,04H[R5]
     MOV.L #00000003H,08H[R5]
     RTS
```

A.6.2 Local Variables and Global Variables

Variables that can be used as local variables must be declared as local variables and not as global variables. There is a possibility that the value of a global variable will be changed by function calling or pointer operations, thus the efficiency of optimization is degraded.

The following advantages are available when local variables are used.

- Access cost is low
- May be assigned to a register
- Efficiency of optimization is good

[Example]

Case in which global variables are used for temporary variables (before improvement) and case in which local variables are used (after improvement)

Source code before improvement

```
int tmp;
void func(int* a, int* b)
{
     tmp = *a;
     *a = *b;
     *b = tmp;
}
```

Assembly-language expansion code before improvement

```
__func:
     MOV.L #_tmp,R4
     MOV.L [R1],[R4]
     MOV.L [R2],[R1]
     MOV.L [R4],[R2]
     RTS
```

Source code after improvement

```
void func(int* a, int* b)
{
     int tmp;
     tmp = *a;
     *a = *b;
     *b = tmp;
}
```

Assembly-language expansion code after improvement

```
__func:
     MOV.L [R1],R5
     MOV.L [R2],[R1]
     MOV.L R5,[R2]
     RTS
```

A.6.3 Offset for Structure Members

A structure member is accessed after adding the offset to the structure address. Since a small offset is advantageous for the size, members often used should be declared at the top.

The most effective combination is within 32 bytes from the top for the **signed char** or **unsigned char** type, within 64 bytes from the top for the **short** or **unsigned short** type, or within 128 bytes from the top for the **int**, **unsigned int**, **long**, or **unsigned long** type.

[Example]

An example in which the code is changed because of the offset of the structure is shown below.

Source code before improvement

```
struct str {
     long L1[8];
     char C1;
};
struct str STR1;
char x;
void func()
{
     x = STR1.C1;
}
```

Assembly-language expansion code before improvement

```
_func:
     MOV.L #_STR1,R4
     MOVU.B 20H[R4],R5
     MOV.L #_x,R4
     MOV.B R5,[R4]
     RTS
```

Source code after improvement

```
struct str {
     char C1;
     long L1[8];
};
struct str STR1;
char x;
void func()
{
     x = STR1.C1;
}
```

Assembly-language expansion code after improvement

```
_func:
     MOV.L #_STR1,R4
     MOVU.B [R4],R5
     MOV.L #_x,R4
     MOV.B R5,[R4]
     RTS
```

Note When defining a structure, declare the members while considering the boundary alignment value. The boundary alignment value of a structure is the most largest boundary alignment value within the structure. The size of a structure becomes a multiple of the boundary alignment value. For this reason, when the end of a structure does not match the boundary alignment value of the structure itself, the size of the structure also includes the unused area that was created for guaranteeing the next boundary alignment.

Source code before improvement

```
/* Boundary alignment value is 4 because the maximum member is the int type */
struct str {
     char C1; /* 1 byte + 3 bytes of boundary alignment */
     long L1; /* 4 bytes */
     char C2; /* 1 byte */
     char C3; /* 1 byte */
     char C4; /* 1 byte + 1 byte of boundary alignment */
}STR1;
```

str size before improvement

```
     .SECTION B,DATA,ALIGN=4
     .glb _STR1
_STR1:      ; static: STR1
     .blkl 3
```

Source code after improvement

```
/* Boundary alignment value is 4 because the maximum member is the int type */
struct str {
     char C1; /* 1 byte */
     char C2; /* 1 byte */
     char C3; /* 1 byte */
     char C4; /* 1 byte */
     long L1; /* 4 bytes */
}STR1;
```

str size after improvement

```
     .SECTION B,DATA,ALIGN=4
     .glb _STR1
_STR1:      ; static: STR1
     .blkl 2
```

A.6.4 Allocating Bit Fields

To set members of different bit fields, the data including the bit field needs to be accessed each time. These accesses can be kept down to one access by collectively allocating the related bit fields to the same structure.

[Example]

An example in which the size is improved by allocating bit fields related to the same structure is shown below.

Source code before improvement

```
struct str
{
     Int flag1:1;
}b1,b2,b3;
void func()
{
     b1.flag1 = 1;
     b2.flag1 = 1;
     b3.flag1 = 1;
}
```

Assembly-language expansion code before improvement

```
_func:
     MOV.L #_b1,R5
     BSET #00H,[R5]
     MOV.L #_b2,R5
     BSET #00H,[R5]
     MOV.L #_b3,R5
     BSET #00H,[R5]
     RTS
```

Source code after improvement

```
struct str
{
     int flag1:1;
     int flag2:1;
     int flag3:1;
}a1;
void func()
{
     a1.flag1 = 1;
     a1.flag2 = 1;
     a1.flag3 = 1;
}
```

Assembly-language expansion code after improvement

```
_func:
     MOV.L #_a1,R4
     MOVU.B [R4],R5
     OR #07H,R5
     MOV.B R5,[R4]
     RTS
```

A.6.5 Optimization of External Variable Accesses when the Base Register is Specified

When **R13** is specified as the base register of the RAM section, accesses to the RAM section are performed relative to the **R13** register. Furthermore, if optimization of inter-module external variable accesses is enabled, the value relative to the **R13** register is optimized, and the instruction size becomes smaller if the value is 8 bits or less.

[Example]

Source code before improvement

```
int a;
int b;
int c;
int d;
void fu{
     a=0;
     b=1;
     c=2;
     d=3;
}
```

Assembly-language expansion code before improvement

```
_func:
     MOV.L #_a,R4
     MOV.L #0000000H,[R4]
     MOV.L #_b,R4
     MOV.L #00000001H,{R4}
     MOV.L #_c,R4
     MOV.L #00000002H,[R4]
     MOV.L #_d,[R4]
     MOV.L #00000003H,[R4]
     RTS
```

Source code after improvement

```
int a;
int b;
int c;
int d;
void fu{
     a=0;
     b=1;
     c=2;
     d=3;
}
```

Assembly-language expansion code after improvement

```
_func:
     MOV.L #0000000H,_a-__RAM_TOP:16[R13]
     MOV.L #0000001H,_b-__RAM_TOP:16[R13]
     MOV.L #0000002H,_c-__RAM_TOP:16[R13]
     MOV.L #0000003H,_d-__RAM_TOP:16[R13]
     RTS
```

A.6.6 Specified Order of Section Addresses by Optimizing Linkage Editor at Optimization of External Variable Accesses

In an instruction that accesses memory in the register relative-address format, the instruction size is small when the displacement value is small.

In some cases, the code size can be improved when the order of allocating the sections by the optimizing linkage editor is changed with reference to the following guidelines.

- Place at the beginning the sections of external variables that are frequently accessed in the function.
- Place at the beginning the sections of external variables with small type sizes.

Note however that the build time gets longer when external variable accesses are optimized because the compiler runs twice.

[Example]

Source code before improvement

```
/* Section D_1 */
char d11=0, d12=0, d13=0, d14=0;
/* Section D_2 */
short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};
/* Section D */
int d41=0, d42=0, d43=0, d44=0, dmy4[60]={0}

void func(int a){
     d11 = a;
     d12 = a;
     d13 = a;
     d14 = a;
     d21 = a;
     d22 = a;
     d23 = a;
     d24 = a;
     d41 = a;
     d42 = a;
     d43 = a;
     d44 = a;
}
```

Assembly-language expansion code before improvement

```
<When the allocation order of sections is "D, D_2, D_1" or D*>
_func:
     MOV.L #d41,R4
     MOV.B R1,0120H[R4]
     MOV.B R1,0121H[R4]
     MOV.B R1,0122H[R4]
     MOV.B R1,0123H[R4]
     MOV.W R1,0100H[R4]
     MOV.W R1,0102H[R4]
     MOV.W R1,0104H[R4]
     MOV.W R1,0106H[R4]
     MOV.L R1,[R4]
     MOV.L R1,04H[R4]
     MOV.L R1,08H[R4]
     MOV.L R1,0CH[R4]
     RTS
```

Source code after improvement

```
/* Section D_1 */
char d11=0, d12=0, d13=0, d14=0;
/* Section D_2 */
short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};
/* Section D */
int d41=0, d42=0, d43=0, d44=0, dmy4[60]={0}

void func(int a){
     d11 = a;
     d12 = a;
     d13 = a;
     d14 = a;
     d21 = a;
     d22 = a;
     d23 = a;
     d24 = a;
     d41 = a;
     d42 = a;
     d43 = a;
     d44 = a;
}
```

Assembly-language expansion code after improvement

```
<When the allocation order of sections is "D_1, D_2, D" or D*>
_func:
     MOV.L #d11,R4
     MOV.B R1,[R4]
     MOV.B R1,01H[R4]
     MOV.B R1,02H[R4]
     MOV.B R1,03H[R4]
     MOV.W R1,04H[R4]
     MOV.W R1,06H[R4]
     MOV.W R1,08H[R4]
     MOV.W R1,0AH[R4]
     MOV.L R1,24H[R4]
     MOV.L R1,28H[R4]
     MOV.L R1,2CH[R4]
     MOV.L R1,30H[R4]
     RTS
```

A.6.7 Interrupt

Due to many registers being saved and restored before and after an interrupt processing, the expected interrupt response time may not be obtained. In such a case, the fast interrupt setting (**fint**) and **fint_register** option should be used to keep down the number of saving and restoring of registers so that the interrupt response time can be reduced.

Note however that usage of the **fint_register** option limits the usable registers in other functions so the efficiency of the entire program is degraded in some cases.

[Example]

Source code before improvement

```
#pragma interrupt int_func
volatile int count;

void  int_func()
{
     count++;
}
```

Assembly-language expansion code before improvement

```
_int_func:
     PUSHM R4-R5
     MOV.L #_count,R4
     MOV.L [R4],R5
     ADD #01H,R5
     MOV.L R5,[R4]
     POPM R4-R5
     RTE
```

Source code after improvement

```
#pragma interrupt int_func(fint)
volatile int count;

void  int_func()
{
     count++;
}
```

Assembly-language expansion code after improvement

```
<When the fint_register=2 option is specified>
_int_func:
     MOV.L #_count,R12
     MOV.L [R12],[R13]
     ADD #01H,R13
     MOV.L R13,[R12]
     RTFI
```

A.7 High-Speed Processing

This section describes high-speed processing.

A.7.1 Loop Control Variable

Loop expansion cannot be optimized if there is a possibility that the size difference prevents the loop control variable from expressing the data to be compared when determining whether the loop end condition is met. For example, if the loop control variable is **signed char** while the data to be compared is **signed long**, loop expansion is not optimized. Thus, compared to **signed char** and **signed short**, it is easier to perform optimization of loop expansion for **signed long**. To optimize loop expansion, specify the loop control variable as a 4-byte integer type.

[Example]

Source code before improvement

```
signed long array_size=16;
signed char array[16];

void func()
{
      signed char I;
      for(i=0;i<array_size;i++)
      {
            array[i]=0;
      }
}
```

Assembly-language expansion code before improvement

```
<When loop=2 is specified>
_func:
      MOV.L #_array_size,R4
      MOV.L [R4],R2
      MOV.L #00000000H,R5
      BRA L11
L12:
      MOV.L #_array,R14
      MOV.L #00000000H,R3
      MOV.B R3,[R5,R4]
      ADD #01H,R5
L11:
      MOV.B R5,R5
      CMP R2,R5
      BLT L12
L13:
      RTS
```

Source code after improvement

```
signed long array_size=16;
signed char array[16];

void func()
{
      signed long I;
      for(i=0;i<array_size;i++)
      {
            array[i]=0;
      }
}
```

Assembly-language expansion code after improvement

```
<When loop=2 is specified>
_func:
      MOV.L #_array_size,R5
      MOV.L [R5],R2
      MOV.L #00000000H,R4
      ADD #0FFFFFFFFH,R2,R3
      CMP R3,R2
      BLE L12
L11:
      MOV.L #_array,R1
      MOV.L R1,R5
      BRA L13
L14:
      MOV.W #0000H,[R5]
      ADD #02H,R5
      ADD #02H,R4
L13:
      CMP R3,R4
      BLT L14
L15:
      CMP R2,R4
      BGE L17 L16:
      MOV.L #00000000H,R5
      MOV.B R5,[R4,R1]
      RTS
L12:
      MOV.L #_array,R5
      MOV.L #00000000H,R3
L19:
      CMPR2,R4
      BGE L17
L20:
      MOV.B R3,[R5+]
      ADD #01H,R4
      BRA L19
L17:
      RTS
```

A.7.2 Function Interface

The number of arguments should be carefully selected so that all arguments can be set in registers (up to four). If there are too many arguments, turn them into a structure and pass the pointer. If the structure itself is passed through and forth, instead of the pointer of the structure, the structure may be too large to be set in a register. When arguments are set in registers, calling and processing at the entry and exit of the function can be simplified. In addition, space in the stack area can be saved. Note that registers **R1** to **R4** are to be used for arguments.

[Example]

Function **f** has four more arguments than the number of registers for arguments.

Source code before improvement

```
void call_func()
{
    func(1,2,3,4,5,6,7,8);
}
```

Assembly-language expansion code before improvement

```
_call_func:
     SUB #04H,R0
     MOV.L #08070605H,[R0]
     MOV.L #00000004H,R4
     MOV.L #00000003H,R3
     MOV.L #00000002H,R2
     MOV.L #00000001H,R1
     BSR _func
     ADD #04H,R0
     RTS
```

Source code after improvement

```
struct str{
     char a;
     char b;
     char c;
     char d;
     char e;
     char f;
     char g;
     char h;
};
struct str arg = {1,2,3,4,5,6,7,8};

void call_func()
{
     func(&arg);
}
```

Assembly-language expansion code after improvement

```
_call_func:
     MOV.L #arg,R1
     BRA _func
```

A.7.3 Reducing the Number of Loops

Loop expansion is especially effective for inner loops. Since the program size is increased by loop expansion, loop expansion should be performed when a fast execution speed is preferred at the expense of the program size.

[Example]

Array **a[]** is initialized.

Source code before improvement

```
extern int a[100];
void func()
{
     int I;
     for( i = 0 ; i < 100 ; i++ ){
          a[i] = 0;
     }
}
```

Assembly-language expansion code before improvement

```
_func:
      MOV.L #00000064H,R4
      MOV.L #_a,R5
      MOV.L #00000000H,R3
L11:
      MOV.L R3,[R5+]
      SUB #01H,R4
      BNE L11
L12:
      RTS
```

Source code after improvement

```
extern int a[100];
void func()
{
      int I;
      for( i = 0 ; i < 100 ; i+=2 )
      {
            a[i] = 0;
            a[i+1] = 0;
      }
}
```

Assembly-language expansion code after improvement

```
_func:
      MOV.L #00000032H,R4
      MOV.L #_a,R5
L11:
      MOV.L #00000000H,[R5]
      MOV.L #00000000H,04H[R5]
      ADD #08H,R5
      SUB #01H,R4
      BNE L11
L12:
      RTS
```

A.7.4 Usage of a Table

If the processing in each **case** label of a **switch** statement is almost the same, consider the usage of a table.
[Example]
The character constant to be assigned to variable **ch** is changed by the value of variable **i**.
Source code before improvement

```
char func(int i)
{
      char ch;
      switch (i) {
            case 0:
                  ch = 'a'; break;
            case 1:
                  ch = 'x'; break;
            case 2:
                  ch = 'b'; break;
      }
      return(ch);
}
```

Assembly-language expansion code before improvement

```
_func:
     CMP #00H,R1
     BEQ L17
L16:
     CMP #01H,R1
     BEQ L19
     CMP #02H,R1
     BEQ L20
     BRA L21
L12:
L17:
     MOV.L #00000061H,R1
     BRA L21
L13:
L19:
     MOV.L #00000078H,R1
     BRA L21
L14:
L20:
     MOV.L #00000062H,R1
L11:
L21:
     MOVU.B R1,R1
     RTS
```

Source code after improvement

```
char chbuf[] = {'a', 'x', 'b'};

char func(int i)
{
     return (chbuf[i]);
}
```

Assembly-language expansion code after improvement

```
_f
     MOV.L #_chbuf,R4
     MOVU.B [R1,R4],R1
     RTS
```

A.7.5 Branch

When comparison is performed in order beginning at the top, such as in an **else if** statement, the execution speed in the cases at the end gets slow if there is many branching. Cases with frequent branching should be placed near the beginning.

[Example]
The return value changes depending on the value of the argument.
Source code before improvement

```
int func(int a)
{
     if (a==1)
          a = 2;
     else if (a==2)
          a = 4;
     else if (a==3)
          a = 0;
     else
          a = 0;
     return(a);
}
```

Assembly-language expansion code before improvement

```
_func:
     CMP #01H,R1
     BEQ L11
L12:
     CMP #02H,R1
     BNE L14
L13:
     MOV.L #00000004H,R1
     RTS
L14:
     CMP #03,R1
     BNE L17
L16:
     MOV.L #00000008H,R1
     RTS
L17:
     MOV.L #00000000H,R1
     RTS
L11:
     MOV.L #00000002H,R1
     RTS
```

Source code after improvement

```
int func(int a)
{
     if (a==3)
          a = 8;
     else if (a==2)
          a = 4;
     else if (a==1)
          a = 2;
     else
          a = 0;
     return (a);
}
```

Assembly-language expansion code after improvement

```
_func:
      CMP #03H,R1
      BEQ L11
L12:
      CMP #02H,R1
      BNE L14
L13:
      MOV.L #00000004H,R1
      RTS
L14:
      CMP #01H,R1
      NE L17
L16:
      MOV.L #00000002H,R1
      RTS
L17:
      MOV.L #00000000H,R1
      RTS
L11:
      MOV.L #00000008H,R1
      RTS
```

A.7.6 Inline Expansion

The execution speed can be improved by performing inline expansion for functions that are frequently called. A significant effect may be obtained by expanding functions that are particularly called in the loop. However, since the program size is inclined to be increased by inline expansion, inline expansion should be performed when a fast execution speed is preferred at the expense of the program size.

[Example]

The elements of array **a** and array **b** are exchanged.

Source code before improvement

```
int x[10], y[10];
static void sub(int *a, int *b, int I)
{
      int temp;
      temp = a[i];
      a[i] = b[i];
      b[i] = temp;
}

void func()
{
      int I;
      for(i=0;i<10;i++)
      {
            sub(x,y,i);
      }
}
```

Assembly-language expansion code before improvement

```
__$sub:
     SHLL #02H,R3
     ADD R3,R1
     MOV.L [R1],R5
     ADD R3,R2
     MOV.L [R2],[R1]
     MOV.L R5,[R2]
     RTS
_func:
     PUSHM R6-R8
     MOV.L #00000000H,R6
     MOV.L #_x,R7
     MOV.L #_y,R8
L12:
     MOV.L R6,R3
     MOV.L R7,R1
     MOV.L R8,R2
     ADD #01H,R6
     BSR __$sub
     CMP #0AH,R6
     BLT L12
L13:
     RTSD #0CH,R6-R8
```

Source code after improvement

```
int x[10], y[10];
#pragma inline(sub)
static void sub(int *a, int *b, int I)
{
     int temp;
     temp = a[i];
     a[i] = b[i];
     b[i] = temp;
}

void func()
{
     int I;
     for(i=0;i<10;i++)
     {
          sub(x,y,i);
     }
}
```

Assembly-language expansion code after improvement

```
; The _sub code was reduced as a result of inline expansion
_func:
     MOV.L #0000000AH,R1
     MOV.L #_y,R2
     MOV.L #_x,R3
L11:
     MOV.L [R3],R4
     MOV.L [R2],R5
     MOV.L R4,[R2+]
     MOV.L R5,[R3+]
     SUB #01H,R1
     BNE L11
L12:
     RTS
```

A.8 Modification of C Source

The language extensions of CC-RX allows its users to create more efficient object files.
Here are two examples describing how to migrate to CC-RX from the other compilers and from CC-RX to the others.
<From other C compiler to the CCRX>

- #pragma
  C source needs to be modified, when C compiler supports the #pragma. Modification methods are examined according to the C compiler specifications.
- Expanded Specifications
  It should be modified when other C compilers are expanding the specifications such as adding keywords etc. Modified methods are examined according to the C compiler specifications.

Note #pragma is one of the pre-processing directives supported by ANSI. The character string next to #pragma is made to be recognized as directives to C compiler. If that directive does not supported by the compiler, #pragma directive is ignored and the compiler continues the process and ends normally.

<From the CCRX to other C compiler>

- The CCRX, either deletes key word or divides # fdef in order shift to other C compiler as key word has been added as expanded function.

Example 1. Disable the keywords

```
#ifndef __RX
#define __evenaccess       /*Change variable with __evenaccess specified to normal
variable*/
#endif
```

Example 2. Change to other keywords

```
#ifdef __RX
_Pragma("inline func")
#else
inline
#endif
void func() { }
```

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Nov 28, 2014	-	First Edition issued
1.01	Sep 14, 2015	14	The description of the license is added.
		21, and others	The following compile options are added. -misra2012 -stack_protector/-stack_protector_all
		21, and others	The following compile options are made usable in the Professional Edition. -misra2004 -ignore_files_misra -check_language_extension
		150, 153	The following description is deleted. - The utf8 option is valid only when the lang=c99 option has been specified.
		156 and others	The assembler option -utf8 is added.
		219	The function of the linker option -crc is expanded.
		324, 335	#pragma stack_protector and #pragma no_stack_protector are added.
		783, and others	Unnecessary messages are deleted.
		785, and others	The following message numbers are added. E0511178 M0523086 W0511179
1.02	Jul 01, 2016	44	The following MISRA-C:2012 rules are added. 2.6 2.7 9.2 9.3 12.1 12.3 12.4 14.4 15.1 15.2 15.3 15.4 15.5 15.6 15.7 16.1 16.2 16.3 16.4 16.5 16.6 16.7 17.1 17.7 18.4 18.5 19.2 20.1 20.2 20.3 20.4 20.5 20.6 20.7 20.8 20.9 20.10 20.11 20.12 20.13 20.14
		220	[Description] is changed.
		272, and others	The library generator option -secure_malloc is added.
		340, and others	Added an alias for each existing intrinsic function, which prefixes "__" to its name.
		344, and others	The intrinsic functions are added. __bclr __bset __bnot
		344, and others	The description of the intrinsic functions is changed.

Rev.	Date	Description	
		Page	Summary
1.02	Jul 01, 2016	617, 618	The following library functions are changed. calloc, free, malloc, realloc
		832, 844	The following message numbers are added. F0523088 W0520171
1.03	Dec 01, 2016	15	The description of "License" is changed.
		15	"Standard and Professional Editions" is added.
		15, 16	"Free Evaluation Editions" is added.
		45	The following MISRA-C:2012 rules are added. 2.2 3.2 5.1 5.6 5.7 5.8 5.9 8.3 8.9 8.14 9.1 9.4 9.5 12.2 17.6 18.7 21.1 21.2 21.3 21.4 21.5 21.6 21.7 21.8 21.9 21.10 21.11 21.12
		63	[Description] is changed.
		73	[Remarks] is changed.
		75	[Remarks] is changed.
		215	[Description] is changed.
		231	[Description] is changed.
		281	[Description] is changed.
		307-315	The descriptions of the following implementation-defined items are changed. 4.1.3 Table 4.1, Table 4.3, Table 4.4, Table 4.5, Table 4.7, Table 4.9, Table 4.10, Table 4.11, Table 4.12, Table 4.13, Table 4.14, Table 4.15, Table 4.16
		343	The description is changed.
		552, 554, 568	The tables are changed.
		574	[Return values] is changed.
		837	F0523073 is changed.
		840	F0563020 is added.
		866, 867	"Standard Library Error Messages" is added.
		875, 876	"V2.06 and Later Versions (Compatibility with V2.05 and earlier)" is added.
1.04	Jun 01, 2017	45	The following MISRA-C:2012 rules are added. 12.5 13.1 13.2 13.5 17.5 17.8 21.13 21.15 21.16
		24, 48, 66	The -avoid_cross_boundary_prefetch option is added.
		57	[Description] is changed.
		58	[Description] is changed.

Rev.	Date	Description	
		Page	Summary
1.04	Jun 01, 2017	198, 207, 214	The -end_record option is added.
		209	-show=relocation_attribute and end_record is added to Specifiable Option.
		234, 235	[Format], [Description] and [Remarks] is changed.
		294	Table 3.2 is changed.
		295	"Linkage Map Information" is changed.
		296	The description of (7) is changed.
		299	The description of (4) is changed.
		319	The description of "Scalar Type (C), Basic Type (C++)" is changed.
		326, 327	The description of "Memory Allocation in Big Endian" is changed.
		346	The description of "Specification of Function for generating a code for detection of stack smashing" is changed.
		796, 844, 851, 864	The following messages are added. C0511200, C0519996, C0519997, C0554098, C0564001, F0563103, W0511180, W0511185, W0561015, W0561016
		851, 863, 864	The following messages are changed. W0511179, W0561004, W0561017
1.05	Dec 01, 2017	23, 28, 38	The -no_warning compile option is added.
		24, 49, 68	The -insert_nop_with_label compile option is added.
		24, 49, 69-71	The -control_flow_integrity compile option is added.
		34	The description of [Default] is added.
		35	The descriptions of [Description] and [Remarks] are changed.
		79	The description of [Default] is added.
		81	The description of [Default] is added.
		204, 213, 229	The -fix_record_length_and_align optimizing linkage editor (rlink) option is added.
		204, 213, 238	The -cfi optimizing linkage editor (rlink) option is added.

Rev.	Date	Description	
		Page	Summary
1.05	Dec 01, 2017	204, 213, 239	The -cfi_add_func optimizing linkage editor (rlink) option is added.
		204, 213, 240	The -cfi_ignore_module optimizing linkage editor (rlink) option is added.
		215	"fix_record_length_and_align" is added to No. 4 of "Specifiable Option".
		215	The description of Notes 3 is changed.
		224	The descriptions of [Description] and [Remarks] are changed.
		228	The entire description is changed.
		244, 245	The "cfi" suboption is added to table 2.17.
		245, 246	The "cfi" suboption is added to the table in [Remarks].
		299	Item No. 30 is added to table 2.19.
		304	Item No. 10 is added to table 3.2.
		310	The section of "CFI Information" is added.
		342	The following macros are added to table 4.22. __STDC_IEC_559__, __STDC_IEC_559_COMPLEX__, and __STDC_ISO_10646__
		430	Item No. 8 is added to table 5.35.
		570-572	The descriptions of definitions in <float.h> are reviewed.
		806, 843, 844, 851-853, 857, 871, 872	The following messages are added. E0511117, E0562366, E0563602, F0563003, F0563150, F0563431, F0563600, F0563601, M0560700, W0561014, and W0561184
		842-844, 851, 852, 857 872, 874	The following messages are changed. E0562311, E0562340, E0562405, E0562417, F0563004, F0563102, M0560005, W0561130, and W0561325
		840-874	The following messages are deleted. E0562021, E0562112, E0562113, E0562203, E0562220, E0562223, E0562224, E0562323, E0562324, E0562402, E0562406, E0562407, E0562408, E0562500, F0563311, F0563312, F0563313, M0560001, M0560512, W0561008, W0561015, W0561180, W0561500, W0561501, and W0561502
		888	The description of Note is changed.

Rev.	Date	Description	
		Page	Summary
1.06	Jun 01, 2018	19	The description of ISA_RX is changed.
		50	[Remarks] is changed.
		52	[Default] is changed.
		53	[Default] is changed.
		94	[Description] and [Remarks] are changed.
		99	[Description] is changed.
		103	[Description] is changed.
		118	[Format], [Description] and [Remarks] are changed.
		120	[Description] is changed.
		146	[Description] and [Remarks] are changed.
		184	[Format], [Description] and [Remarks] are changed.
		204, 213, 237	The -split_vect optimizing linkage editor (rlink) option is added.
		215	The following specifiable options are deleted. padding, vectn, vect, and split_vect
		222	[Format], [Description] and [Remarks] are changed.
		232	[Remarks] is changed.
		235	[Description] is changed.
		241	[Format], [Description] and [Example] are changed.
		319	The description of "Function definitions - storage of formal parameters" is changed.
		343, 344	The following macros are added to table 4.22. __RXV3 and __RX_ISA_VERSION__
		349, 350	The description of "Interrupt Function Creation" is changed.
		385	The descriptions of [Remarks] of set_extb and get_extb are changed.
		410	The description of .RVECTOR is changed.
		432	The following macros are added to table 5.35. __RXV3 and __RX_ISA_VERSION__
		434	The description of "Instructions" is changed.
		435, 436	The description of "Register Configuration" is changed.
		466-581	The following instructions are added. BFMOV, BFMOVZ, EMACA, EMSBA, EMULA, FSQRT, FTOU, MACLH, MOVCO, MOVLI, MSBHI, MSBLH, MSBLO, MULLH, MVFACGU, MVFACLO, MVTACGU, RACL, RDACL, RDACW, RSTR, SAVE, and UTOF
		488-585	The following instructions are changed. FADD, FMUL, FSUB, MACHI, MACLO, MULHI, MULLO, MVFACHI, MVFACMI, MVFC, MVTACHI, MVTACLO, MVTC, POPC, PUSHC, RACW, RTE, RTFI, STNZ, STZ, and XOR

Rev.	Date	Description	
		Page	Summary
1.06	Jun 01, 2018	587	Table 6.1 is changed.
		597, 598	The following reentrants are added. fabs, fabsf, and fabsl
		652	Table 7.10 is changed.
		653-655	Table 7.11 is changed.
		792	The description of "Overview" is changed.
		792	The description of Notes 1 is changed.
		796	The description of "Coding Example of Initial Setting Routine" is changed.
		796	The description of "Low-Level Interface Routines" is changed.
		813	The description of "Example of Low-Level Interface Routine for Reentrant Library" is changed.
		817	The description of Notes 1 is changed.
		818	The description of "resetprg.c: Initial Setting Routine (Reset Vector Function)" is changed.
		820	[Reference] is changed.
		875, 912, 914	The following messages are added. E0552011, W0561143, and W0561331
		848, 875, 884, 889	The following messages are changed. E0511154, E0552010, E0562410, and F0523073
		857, 889	The following messages are deleted. E0520412, F0523054, F0523055, and F0523056
		928	"Version of Compiler Package" is added.
1.07	Dec 01, 2018	23, 29, 42	The -truncated_address_initializer compile option is added.
		23, 29, 51	The -misra_intermodule compile option is added.
		24, 65	The description of the -nouse_div_inst compile option is changed.
		26, 110	The description of the -fpu compile option is changed.
		26, 111	The description of the -nofpu compile option is changed.
		26, 79, 112	The -dpfpu compile option is added.

Rev.	Date	Description	
		Page	Summary
1.07	Dec 01, 2018	26, 79, 113	The -nodpfpu compile option is added.
		58	[Description] of the compile option -stuff is changed.
		97	[Description] of the compile option -library is changed.
		128	[Default] of the compile option -dbl_size is changed.
		143	[Description] of the compile option -base is changed.
		147	[Default] and [Description] of the compile option -pid are changed.
		168, 173	The description of the -chkpm assembler command option is changed.
		168, 174	The description of the -chkfpu assembler command option is changed.
		168, 175	The description of the -chkdsp assembler command option is changed.
		168, 170, 176	The -chkdpfpu assembler command option is added.
		168, 182	The description of the -fpu assembler command option is changed.
		168, 183	The description of the -nofpu assembler command option is changed.
		168, 177, 184	The -dpfpu assembler command option is added.
		168, 177, 185	The -nodpfpu assembler command option is added.
		168, 177, 186	The -bank assembler command option is added.
		168, 177, 187	The -nobank assembler command option is added.
		195	[Remarks] of the assembler command option -cpu is changed.
		215, 225, 275, 282	The -lib_rename optimizing linkage editor option is added.
		311	The following options are added to table 2.19. create_unfilled_area, stack_protector, stack_protector_all, misra2004, misra2012, and misra_intermodule
		335	Table 4.7 is changed.
		356	The following macro is added to table 4.22. __DPFPU

Rev.	Date	Description	
		Page	Summary
1.07	Dec 01, 2018	362, 363, 365	The description of "Interrupt Function Creation" is changed.
		377, 400-402	The following Intrinsic Functions are added. __set_dpsw, __get_dpsw, __set_decnt, __get_decnt, __set_depc, and __get_depc
		404	The description of "ASSEMBLY LANGUAGE SPECIFICATIONS" is changed.
		407	The description of "General Instruction Addressing" is changed.
		410	The description of "Specific Instruction Addressing" is changed.
		448	The following macro is added to table 5.35. __DPFPU
		449	"Instructions" is deleted.
		449, 450	The following sections are added to table 6.1. C_8, D_8, and B_8
		658	The description of "Initialization of Control Registers" is changed.
		661	"Coding Example of Initial Setting Routine" is changed.
		683, 684	"resetprg.c: Initial Setting Routine (Reset Vector Function)" is changed.
		695	The description of "Application Code Generation (pic and pid Options)" is changed.
		715, 764, 775	The following messages are added. E0520069, W0520070, and W0551017
		738, 764	The following messages are changed. E0523065 and W0520069
1.08	Apr 01, 2019	26, 79, 114, 312	The -tfu compile option is added.
		357	The following macros are added to table 4.22. __TFU and __TFU_MATHLIB
		367	Examples 7 is added.
		379, 408	The __set_depc intrinsic function is deleted.
		379, 409, 410	The following intrinsic functions are added. __init_tfu, __sincosf, and __atan2hypotf
		406-408	The descriptions of [Remarks] of __set_dpsw, __get_dpsw, __set_decnt, __get_decnt, and __get_depc are changed.
		469, 470	The following "Reentrant" are changed. atan2, cos, sin, fabs, atan2f, cosf, sinf, and fabsl
		471	Notes 2 is added.
		471	The following "Reentrant" are changed. atan2l, cosl, sinl, sqrtl, and fabsl

Rev.	Date	Description	
		Page	Summary
1.08	Apr 01, 2019	475	Notes 1 and Notes 2 are added.
		490, 503	The descriptions of the hypot, hypotf, and hypotl library function are changed.
		557	The descriptions of [Format] of bsearch and qsort are changed.
		680	"Initializing the Trigonometric Function Unit" is added.
		683	"Coding Example of Initial Setting Routine" is changed.
		760, 772, 777	The following messages are added. E0523129, E0562600, and F0523129
1.09	Nov 01, 2019	12	The description of "Copyrights" is changed.
		47	For the -misra2012 compile option, [Remarks] is changed and [Example] is added.
		47	The following MISRA-C:2012 rules are added: 8.13, 14.2, 14.3
		23, 51, 56	The -g_line compile option is added.
		114	[Description] of the tfu compile option is changed.
		214, 217, 224, 227, 282	The ALLOW_DUPLICATE_MODULE_NAME optimizing linkage editor (rlink) option is added
		313	Table 2.19 is changed.
		334	"Implementation Defined Items" is deleted, and "Implementation-defined behavior of C90" is added.
		339	"Implementation-defined behavior of C99" is added.
		369, 370	The description of "Section Switch" is changed.
		513	The descriptions of functions are changed.
		569	HUGE_VAL and HUGE_VALL are added to "HUGE_VALF" in the "Definition Name" row of the table.
		702	In "cacosf / cacos / cacosl", [Return values] and [Remarks] are changed.
		703	In "casinf / casin / casinl", [Return values] and [Remarks] are changed.
		704	In "catanf / catan / catanl", [Return values] is changed.
		708	In "cacoshf / cacosh / cacoshl", [Return values] and [Remarks] are changed.
		740	The description of maxdiv is entirely changed.

Rev.	Date	Description	
		Page	Summary
1.10	Nov 01, 2020	Front cover	The target CPU cores are added.
		18	The tool usage information file is added to Table 2.1.
		25, 79, 84, 118	The -branch_chaining compile option is added.
		25, 79, 82, 83, 85, 119	The -nobranch_chaining compile option is added.
		31	[Description] of the -preinclude compile option is changed.
		131	[Description] of the -dbl_size compile option is changed.
		218, 281	A typographical error is corrected (stack information file).
		218, 279, 295	The -verbose optimizing linkage editor (rlink) option is added.
		922	The following registers are added to Table 9.1: DCMR, DPSW, DECNT, and DEPC
		960	The following message is added: E0552020
		1015	Notes are added.

Rev.	Date	Description	
		Page	Summary
1.11	Nov 01, 2021	10	The description of "GENERAL" is changed.
		18	The description of "Environment Variables" is changed.
		24, 79, 91	The -type_size_access_to_volatile compile option is added.
		82	[Remarks] of the -speed compile option is changed.
		83	[Remarks] of the -size compile option is changed.
		85	[Remarks] of the -inline compile option is changed.
		86	[Remarks] of the -noinline compile option is changed.
		101	[Description] of the -map compile option is changed.
		138	The description of the -pack compile option is entirely changed.
		139	The description of the -unpack compile option is entirely changed.
		215, 218, 225	The following option name is changed to lowercase. -allow_duplicate_module_name
		262	[Remarks] of the -optimize optimizing linkage editor (rlink) option is changed.
		344	The description in (47) in "Implementation-defined behavior of C99" is changed.
		368	A description is added in "#pragma Directive".
		377, 378, 382	Descriptions are changed in "Using Extended Specifications".
		385	The description is changed in "Using a Keyword".
		468	The description is changed in "List of Section Names".
		528	The specification format of frexp is changed.
		675, 677, 679, 742	The descriptions of the following functions are changed. div, ldiv, lldiv, imaxdiv
		931, 977, 982, 995	The following messages are changed: E0511178, F0563430, W0511180, W0511185, W0561016, W0561017
		935, 987	Errors in the following messages are corrected: E0520137, W0521053
		952	The following message is deleted: E0521212
		979	The following messages are added: M0520177, M0520550, M0520826
		1018	The descriptions of "Performing In-Line Expansion of Functions" is changed.
		1019	The descriptions of "Using CPU Instructions in C Language" is changed.
		1038	The descriptions of "Modification of C Source" is changed.

Rev.	Date	Description	
		Page	Summary
1.12	Dec 01, 2022	25, 79, 115	The -tfu_version compile option is added.
		25, 79, 116	The -nosave_tfu compile option is added.
		66, 193, 258, 310, 370, 477, and others	Applicable revisions are added.
		113, 114	[Description] and [Remarks] of the -tfu compile option are changed.
		156	[Remarks] of the -save_acc compile option is changed. MACW instruction -> MACLO instruction
		226	[Remarks] of the -entry optimizing linkage editor (rlink) option is changed.
		247, 248	[Description] and [Example] of the -crc optimizing linkage editor (rlink) option are changed.
		318	The following options are added to the "Options that Become Invalid" column in Table 2.19. tfu_version, nosave_tfu
		339	The description of (23) in "Undefined Behavior" is changed.
		341	The description of (25) in "Implementation-defined behavior of C90" is changed.
		355	The information in the "Signed/Unsigned" column for the "Pointer to a data member" row in Table 4.1 is changed.
		370	The "Predefined Macro" column in Table 4.6 is changed.
		377, 378	The following interrupt specifications are added to the description of "Interrupt Function Creation". tfu, no_tfu
		395, 428-431	The following intrinsic functions are added. __sincosfx, __sinfx, __cosfx, __atan2hypotfx, __atan2fx, __hypotfx
		425	[Description], [Example], and [Remarks] of the __init_tfu intrinsic function are changed.
		426	[Example] and [Remarks] of the __sincosf intrinsic function are changed.
		427	[Example] and [Remarks] of the __atan2hypotf intrinsic function are changed.
		492, 496	Notes of Tables 7.4 and 7.5 are changed.

Rev.	Date	Description	
		Page	Summary
1.12	Dec 01, 2022	757, 758	The "Description" column of the table for <stdint.h> is changed.
		889	The coding example in "Coding Example of Initial Setting Routine" is changed.
		966, 976, 981, 986	The following messages are added. E0523049, E0562220, F0520571, F0563115
		1005	The following message is changed. W0561017
1.13	Dec 01, 2023	10	The description of "Overview" is changed.
		14	Table 1.2 and its note are changed.
		16	The description of "Preprocessor" in "Overview" is changed.
		18	The "Default When Specification is Omitted" column in Table 2.2 is changed.
		32	[Description] of the -define compile option is changed.
		40	[Description] of the -comment compile option is changed.
		50	[Remarks] of the -misra_intermodule compile option is changed.
		61	[Description] of the -instalign4 compile option is changed.
		62	[Description] of the -instalign8 compile option is changed.
		76, 78	The section number is added to the reference title.
		97	[Default] of the -scope compile option is changed.
		98	[Default] of the -noscope compile option is changed.
		101	[Default] of the -map compile option is changed.
		104	[Default] of the -nomap compile option is changed.
		220, 265, 273	The -ALLOW_OPTIMIZE_ENTRY_BLOCK optimizing linkage editor (rlink) option is added.
		238	[Description] of the -rom optimizing linkage editor (rlink) option is changed.
		248	[Description] of the -crc optimizing linkage editor (rlink) option is changed.
		251	[Description] of the -padding optimizing linkage editor (rlink) option is changed.
		298	[Remarks] of the -total_size optimizing linkage editor (rlink) option is changed.
		307	Descriptions are added in "Library Generator Options".
		319, 320	Table 2.19 is changed.

Rev.	Date	Description	
		Page	Summary
1.13	Dec 01, 2023	323	The table in "Object Information" is changed.
		343	The description of (33) in "Implementation-defined behavior of C90" is changed.
		371, 372	Table 4.6 and its note are changed.
		387	Descriptions are changed in "Alignment Value Specification for Structure Members and Class Members" and "Allocation of a Variable to the Absolute Address" in "Using Extended Specifications".
		397, 428-430, 432	The parameter names are changed as follows: sin → s, cos → c, atan2 → a, and hypot → h
		478, 479	Table 5.35 and its note are changed.
		492	Table 7.4 is changed.
		494-497	Table 7.5 is changed.
		524	The lines for FP_FAST_FMA, FP_FAST_FMAF, and FP_FAST_FMAFL are deleted from the table in "<math.h>".
		529	[Remarks] of the acos, acosf, and acosl functions is changed.
		530	[Remarks] of the asin, asinf, and asinl functions is changed.
		531	[Remarks] of the atan, atanf, and atanl functions is changed.
		532	[Remarks] of the atan2, atan2f, and atan2l functions is changed.
		535	[Remarks] of the tan, tanf, and tanl functions is added.
		626, 912, 934	The table number of the reference title is changed.
		887	The description of "Overview" in "STARTUP" is changed.
		941, 959, 996	The following messages are changed: C0519996, E0520791, and W0511164
		941, 942, 978, 991, 992, 1013	The following messages are added: C0520000, C0529000, C0590001, E0562114, F0593021, F0593300, F0593302, F0593303, F0593305, F0593320, F0593321, F0593322, F0593324, F0593325, F0593326, F0593327, W0591300, and W0591301
		944	The following message is deleted: E0511165
		1026	The description of "V2.06 and Later Versions (Compatibility with V2.05 and earlier)" is changed.
		1030	The description of "Performing const Declaration for Variables with Unchangeable Initialized Data" is changed.

CC-RX User's Manual

Publication Date: Rev.1.00 Nov 28, 2014
Rev.1.13 Dec 01, 2023

Published by: Renesas Electronics Corporation

CC-RX

Renesas Electronics Corporation

R20UT3248EJ0113