User's Manual

RX Family C/C++ Compiler, Assembler, Optimizing Linkage Editor

Compiler Package V.1.01 User's Manual

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

Renesas Electronics
www.renesas.com

Rev.1.00 Feb 2011

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

Preface

This manual explains how to use the C/C++ compiler, assembler, and optimizing linkage editor for the RX family microcomputers. Please read this manual before using this system to fully understand the system. This system translates source programs written in C/C++ language or assembly source programs into relocatable and absolute object programs for the RX family microcomputers.

This manual is intended for an IBM PC*[1] compatible machine and Microsoft® Windows® XP operating system, Microsoft® Windows® Vista operating system, or Microsoft® Windows 7® operating system*[2] that runs on other compatible machines.

Notes on Symbols: The following symbols are used in this manual.

Symbols Used in This Manual

Symbol	Explanation
< >	Indicates an item to be specified.
[ ]	Indicates an item that can be omitted.
...	Indicates that the preceding item can be repeated.
Δ	Indicates one or more blanks.
|	Indicates that one of the items must be selected.

Notes: 1. IBM is a registered trademark of International Business Machines Corporation

2. Microsoft® and Windows® are registered trademarks of Microsoft Corporation in the United States and other countries.

* All other company names and product names are trademarks or registered trademarks of corresponding companies.

Contents

Section 1 Overview 1
1.1 Configuration of Compiler 1
1.1.1 Compile Driver Input 2
1.1.2 Compile Driver Output 2
1.1.3 ccrx 2
1.1.4 asrx 2
1.1.5 optlnk 2
1.1.6 lbgrx 3
1.2 Option Specification Rules 3
1.2.1 Compiler (ccrx) 3
1.2.2 Assembler (asrx) 4
1.2.3 Optimizing Linkage Editor (optlnk) 4
1.2.4 Library Generator (lbgrx) 4
1.3 Command Description Examples 5
1.3.1 Compilation, Assemble, and Linkage by One Command 5
1.3.2 Compilation and Assemble by One Command 5
1.3.3 Compilation, Assemble, and Linkage by Separate Commands 6
1.3.4 Assemble and Linkage by One Command 7
1.3.5 Assemble and Linkage by Separate Commands 7

Section 2 C/C++ Compiler Options 9
2.1 Source Options 9
2.2 Object Options 20
2.3 List Options 30
2.4 Optimize Options 33
2.5 Microcontroller Options 56
2.6 Assemble and Linkage Options 78
2.7 Other Options 81

Section 3 Library Generator Options 85
3.1 Library Generator Options 85
3.2 Compiler Options that Become Invalid 90

Section 4 Assembler Options 93
4.1 Source Options 93
4.2 Object Options 96

4.3 List Options 98
4.4 Microcontroller Options 100
4.5 Other Options 107

Section 5 Optimizing Linkage Editor Options 109
5.1 Option Specifications 109
5.1.1 Command Line Format 109
5.1.2 Subcommand File Format 109
5.2 List of Options 109
5.2.1 Input Options 110
5.2.2 Output Options 117
5.2.3 List Options 142
5.2.4 Optimize Options 146
5.2.5 Section Options 156
5.2.6 Verify Options 161
5.2.7 Other Options 166
5.2.8 Subcommand File Options 178
5.2.9 CPU Option 180
5.2.10 Options Other Than Above 181

Section 6 Environment Variables 185
6.1 Environment Variables 185
6.2 Predefined Macros 187

Section 7 File Specifications 189
7.1 Naming Files 189
7.2 Source List 191
7.2.1 Structure of Source List 191
7.2.2 Source Information 191
7.2.3 Object Information 191
7.2.4 Statistics Information 194
7.2.5 Compiler Command Specification Information 194
7.2.6 Assembler Command Specification Information 195
7.3 Linkage List 196
7.3.1 Structure of Linkage List 196
7.3.2 Option Information 198
7.3.3 Error Information 198
7.3.4 Linkage Map Information 199
7.3.5 Symbol Information 200
7.3.6 Symbol Deletion Optimization Information 201

7.3.7 Cross-Reference Information 202
7.3.8 Total Section Size 203
7.3.9 Vector Information 203
7.3.10 CRC Information 204
7.4 Library List 205
7.4.1 Structure of Library List 205
7.4.2 Option Information 206
7.4.3 Error Information 206
7.4.4 Library Information 207
7.4.5 Module, Section, and Symbol Information within Library 208

Section 8 Programming 209
8.1 Program Structure 209
8.1.1 Sections 209
8.1.2 C/C++ Program Sections 209
8.1.3 Assembly Program Sections 213
8.1.4 Linking Sections 215
8.2 Function Calling Interface 219
8.2.1 Rules Concerning the Stack 219
8.2.2 Rules Concerning Registers 220
8.2.3 Rules Concerning Setting and Referencing Parameters 223
8.2.4 Rules Concerning Setting and Referencing Return Values 226
8.2.5 Examples of Parameter Allocation 228
8.2.6 Method for Mutual Referencing of External Names 232
8.3 Startup Program Creation 234
8.3.1 Fixed Vector Table Setting 235
8.3.2 Initial Setting 235
8.3.3 Coding Example of Initial Setting Routine 240
8.3.4 Low-Level Interface Routines 241
8.3.5 Termination Processing Routine 259
8.3.6 Startup Program Created in Integrated Development Environment 263
8.4 Usage of PIC/PID Function 280
8.4.1 Terms Used in this Section 280
8.4.2 Function of Each Option 280
8.4.3 Restrictions on Applications 282
8.4.4 System Dependent Processing Necessary for PIC/PID Function 282
8.4.5 Combinations of Code Generating Options 283
8.4.6 Master Startup 285
8.4.7 Application Startup 287

Section 9 C/C++ Language Specifications 293
9.1 Language Specifications 293
9.1.1 Compiler Specifications 293
9.1.2 Internal Data Representation 301
9.1.3 Floating-Point Number Specifications 319
9.1.4 Operator Evaluation Order 327
9.1.5 Conforming Language Specifications 328
9.2 Extended Specifications 329
9.2.1 #pragma Extension Specifiers and Keywords 329
9.2.2 Intrinsic Functions 359
9.2.3 Section Address Operators 397
9.3 C/C++ Libraries 400
9.3.1 Standard C Libraries 400
9.3.2 EC++ Class Libraries 675
9.3.3 Reentrant Library 759
9.3.4 Unsupported Libraries 763

Section 10 Assembly Language Specifications 765
10.1 Coding Rules 765
10.1.1 Reserved Words 765
10.1.2 Names 765
10.1.3 Mnemonic Line Format 766
10.1.4 Coding of Labels 767
10.1.5 Coding of Operation 767
10.1.6 Coding of Operands 771
10.1.7 Coding of Comments 783
10.2 Optimum Instruction Selection 784
10.2.1 Selection of Optimum Instruction Format 784
10.2.2 Selection of Optimum Branch Instruction 795
10.3 Assembler Directive Coding 798
10.3.1 Address Directives 798
10.3.2 Assembler Directives 814
10.3.3 Link Directives 817
10.3.4 Source List Directive 821
10.3.5 Conditional Assembly Directives 822
10.3.6 Extended Function Directives 825
10.3.7 Macro Directives 832
10.3.8 Specific Compiler Directives 846

Section 11 Compiler Error Messages 847

11.1 Error Format and Error Levels 847
11.2 List of Messages 847
11.3 Standard Library Error Messages 926

Section 12 Assembler Error Messages 929
12.1 Error Format and Error Levels 929
12.2 List of Messages 929

Section 13 Error Messages for the Optimizing Linkage Editor 943
13.1 Error Format and Error Levels 943
13.2 Return Values for Errors 943
13.3 List of Messages 944

Section 14 Translation Limits 963
14.1 Translation Limits of Compiler 963
14.2 Translation Limits of Assembler 965

Section 15 Usage Notes 967
15.1 Notes on Program Coding 967
15.2 Notes on Compiling a C Program with the C++ Compiler 973
15.3 Notes on Options 974
15.4 Compatibility with an Older Version or Older Revision 975
15.4.1 Compatibility with V.1.00 975

Section 16 Appendix 979
16.1 S-Type and HEX File Formats 979
16.1.1 S-Type File Format 979
16.1.2 HEX File Format 981
16.2 ASCII Code List 984

Section 1 Overview

1.1 Configuration of Compiler

The configuration of the C/C++ compiler for the RX family is shown below.

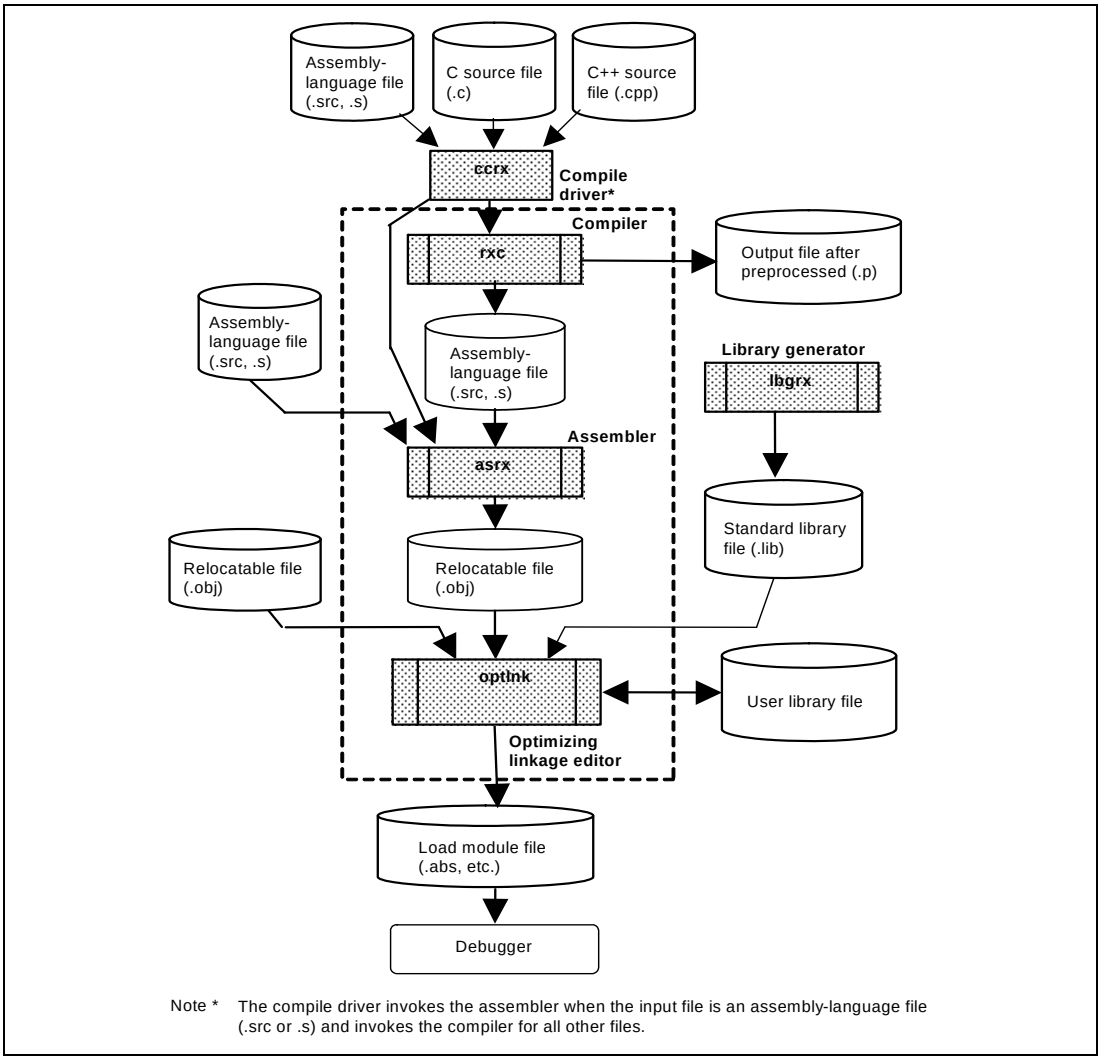


Figure 1.1 Configuration of Compiler

1.1.1 Compile Driver Input

The files that can be input to the compile driver are assembly-language files (.src, .s) and source files (.c, .cpp) which are written in the C language conforming to the ANSI standard (C89/C99 (except for variable-length arrays)), the C++ language conforming to the ANSI standard, or the EC++ language, consisting of ASCII characters and shift JIS characters (can be changed to EUC, Latin1, or UTF-8 by options).

1.1.2 Compile Driver Output

The files output from the compile driver are output files after preprocessed (.p), assembly-language files (.src, .s), relocatable files, and load module files.

1.1.3 ccrx

ccrx is an executable file of the compile driver.

ccrx can be used to execute the processing from compilation to linkage at once by specifying options. In addition, following the **ccrx** startup options **-asmcmd**, **-lnkcmd**, **-asmopt**, and **-lnkopt**, the assembler (**asrx**) options and optimizing linkage editor (**optlnk**) options can be specified.

1.1.4 asrx

asrx is an executable file of the assembler.

asrx is used to convert assembly-language files (.src, .s) into relocatable files.

1.1.5 optlnk

optlnk is an executable file of the optimizing linkage editor.

optlnk is used to convert multiple relocatable files (.obj) and library files (.lib) into load module files (.abs, etc.) or library files (.lib).

1.1.6 lbgrx

lbgrx is an executable file of the library generator.

lbgrx is used to generate standard library files (.lib) according to the options specified by the user.

1.2 Option Specification Rules

The following describes the startup commands for this compiler package.

Before using these commands, refer to section 6, Environment Variables, and check that the required environment variables have been set.

1.2.1 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).

[Command description format]

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

1.2.2 Assembler (asrx)

asrx is the startup command for the assembler.

[Command description format]

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

1.2.3 Optimizing Linkage Editor (optlnk)

optlnk 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]

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

1.2.4 Library Generator (lbgrx)

lbgrx is the startup command for the library generator.

[Command description format]

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

1.3 Command Description Examples

1.3.1 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 **optlnk** to generate an absolute file (tp.abs).

[Command description]

```
ccrx -cpu=rx600 -output=abs=tp.abs tp1.c tp2.c
```

[Remarks]

- 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.
- 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.
- 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.
- 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.

1.3.2 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 -cpu=rx600 -output=obj tp1.c tp2.c

optlnk -form=abs -output=tp.abs -subcommand=cmd.sub tp1.obj tp2.obj
```

[Remarks]

- When the **-output=obj** option is specified in **ccrx**, **ccrx** generates relocatable files.
- In order to change relocatable file names, their C/C++ source files have to be input in **ccrx**, one file each.
- When the **form** option in **optlnk** is changed to **-form=sty**, the file after linkage will be generated as a Motorola S type file.

1.3.3 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 **optlnk** to generate an absolute file (tp.abs).

[Command description]

```
ccrx -cpu=rx600 -output=src tp1.c tp2.c

asrx tp1.src tp2.src

optlnk -form=abs -output=tp.abs -subcommand=cmd.sub tp1.obj tp2.obj
```

[Remarks]

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

1.3.4 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 **optlnk** to generate an absolute file (tp.abs).

[Command description]

```
ccrx -cpu=rx600 -output=abs=tp.abs tp1.src tp2.src
```

[Remarks]

- 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.

1.3.5 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 **optlnk** to generate an absolute file (tp.abs).

[Command description 1]

```
ccrx -cpu=rx600 -output=obj tp1.src tp2.src
optlnk -form=abs -output=tp.abs -subcommand=cmd.sub tp1.obj tp2.obj
```

[Command description 2]

```
asrx -cpu=rx600 tp1.src tp2.src
optlnk -form=abs -output=tp.abs -subcommand=cmd.sub tp1.obj tp2.obj
```

Section 2 C/C++ Compiler Options

2.1 Source Options

Table 2.1 Source Options

No.	Option	Dialog Menu	Description
1	lang = { c | cpp | ecpp | c99 }	C/C++ <Source> [Show entries for:] [Source file] [Language :] [C :] [C (C89)] [C99] [C++ :] [C++] [EC++]	Compiles as a C (C89) source file. Compiles as a C++ source file. Compiles as an EC++ source file. Compiles as a C (C99) source file.
2	include = <path name>[,…]	C/C++ <Source> [Show entries for :] [Include file directories]	Specifies the name of the path to the folder that stores the include file.
3	preinclude = <file name>[,…]	C/C++ <Source> [Show entries for :] [Preinclude files]	Includes the specified files at the head of compiling units.
4	define = <sub>[,…] <sub>:<macro name>[=<string>]	C/C++ <Source> [Show entries for :] [Defines]	Defines <string> as <macro name>.
5	undefine = <sub>[,…] <sub>:< macro name >	C/C++ <Source> [Show entries for :] [Undefines]	Disables the predefined macro of <macro name>.

No.	Option	Dialog Menu	Description
6	message nomessage[=<error number> [-<error number>][,…]]	C/C++ <Source> [Show entries for :] [Messages] [Repressed information level messages]	Enables information message output. Disables information message output.
7	change_message =<sub>[,…] <sub>:<level> [=<n>[-m][,…]] <level>:{Information | warning | error }	C/C++ <Other> [User defined options :]	Changes the level of the compiler output message.
8	file_inline_path=< path name>[,…]	C/C++ <Source> [Show entries for :] [File inline path]	Specifies the name of the path to the folder that stores a file for inter-file inline expansion.
9	comment = { nest | nonest }	C/C++ <Source> [Show entries for:] [Source file] [Allow comment nest]	Permits comment (/* */) nesting. Does not permit comment (/* */) nesting.
10	check={ nc | ch38 | shc }	C/C++ <Source> [Show entries for:] [Source file] [Interchangeability check :] [None] [NC compiler] [H8 compiler] [SH compiler]	Checks the compatibility with an existing program.

lang

Format: lang= { c | cpp | ecpp | c99 }

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.

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.

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.

include

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 (,).

The system include file is searched for in the order of the folders specified by the **include** option, the folders specified by environment variable **INC_RX**, and the folders specified by environment variable **BIN_RX**.

The user include file is searched for in the order of the folders containing source files to be compiled, the folders specified by the **include** option, the folders specified by environment variable **INC_RX**, and the folders specified by environment variable **BIN_RX**.

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

preinclude

Format: preinclude=<file name>[,...]

Description: This option includes the specified file contents at the head of the compiling unit. Multiple file names can be specified by separating them with a comma (,).

If there is more than one folder specified by the **include** option, search is performed in turn starting from the leftmost folder.

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

define

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. 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

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: For the specifiable predefined macros, refer to section 6.2, Predefined Macros.

If this option is specified for more than one time, all specified macro names will be undefined.

message, nomessage

Format: message

nomessage [= <error number> [- <error number>][,…]

Description: These options specify whether or not the information-level messages are output.

When the **message** option is specified, information-level messages are output.

When the **nomessage** option is specified, output of the information-level messages is disabled. When an error number is specified as a suboption, the output of the specified information-level message will be disabled. Multiple error numbers can be specified by separating them with a comma (,).

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>.

The default for these options is **nomessage**.

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.

change_message

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 (,).

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.

The level of error messages cannot be controlled by this option.

file_inline_path

Format: file_inline_path=<path name>[,…]

Description: This option specifies the name of the path where a file for inter-file inline expansion is stored.

Multiple path names can be specified by separating them with a comma (,). The file for inter-file inline expansion is searched for in the order of the folders specified by the **file_inline_path** option and the current folder.

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

comment

Format: comment = { nest | nonest }

Description: When **comment=nest** is specified, nested comments are allowed to be written in the source file.

When **comment=nonest** is specified, writing nested comments will generate an error.

The default for this option is **comment=nonest**.

Example:
```
/* This is an example of /* nested */ comment */
                                        ↑
                                       (1)
```
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.

check

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 **cha**r 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 section 9.1.2, Internal Data Representation.

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.

2.2 Object Options

Table 2.2 Object Options

No.	Option	Dialog Menu	Description
1	output	C/C++	Specifies the output file type.
	= {prep	<Object>	Outputs a source file after preprocessed.
		[Output file type :]	
	| src	[Machine code]	Outputs an assembly-language file.
		[Assembly source code]	
	| obj	[Preprocessed source file]	Outputs a relocatable file.
	| abs		Outputs an absolute file.
	| hex		Outputs an Intel hex type file.
	| sty} [= file name]		Outputs a Motorola S type file.
2	noline	C/C++	Disables **#line** output at preprocessor expansion.
		<Object>	
		[Output file type :]	
		[Suppress #line in preprocessed source file]	
3	debug	C/C++	Outputs debugging information.
		<Object>	
	nodebug	[Generate debug information]	Does not output debugging information.
4	section = <sub>[,…]	C/C++	Changes the section name.
		<Object>	
	<sub>:	[Details]	
	{P = <section name>	[Section :]	Section name of program area
	| C = <section name>	[Program section (P)]	Section name of constant area
	| D = <section name>	[Const section (C)]	Section name of initialized data area
		[Data section (D)]	
	| B = <section name>	[Bss section (B)]	Section name of uninitialized data area
		[Literal section (L)]	
	| L = <section name>	[Switch table section (W)]	Section name of literal area
	| W = <section name>}		Section name of **switch** statement branch table area

No.	Option	Dialog Menu	Description
5	stuff	C/C++ <Object> [Details] [Disposition of variables :]	Allocates variables to sections matching the alignment value.
	nostuff[= { B	[Bss section (B)]	Allocates uninitialized variables to 4-byte boundary alignment sections.
	| D	[Data section (D)]	Allocates initialized variables to 4-byte boundary alignment sections.
	| C	[Const section (C)]	Allocates **const** qualified variables to 4-byte boundary alignment sections.
	| W } [,…]]	[Switch table section (W)]	Allocates **switch** statement branch tables to 4-byte boundary alignment sections.
6	-instalign4[=<sub>]	C/C++ [Adjustment for instruction in branch:]	Aligns instructions at branch destinations to 4-byte boundaries.
	-instalign8[=<sub>]	[instalign4] [instalign8]	Aligns instructions at branch destinations to 8-byte boundaries.
	-noinstalign	[none] [none] [loop] [inmostloop]	Does not align instructions at branch destinations. Head of loop Head of inmost loop
	<sub>: { loop | inmostloop }		

output

Format: output = <sub> [=<file name>]

<sub>: { prep | src | obj | abs | hex | sty }

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.

The default for this option is **output=obj**.

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 stored in the specified folder; however, when no folder has been specified, the intermediate file is stored in the current folder.

noline

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, nodebug

Format: debug

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.

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

The default for these options is **nodebug**.

section

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.

The default for this option is **section=P=P,C=C,D=D,B=B,L=L,W=W**.

Remarks: For details on correspondence between programs and section names, refer to section 8.1.2, C/C++ Program Sections.

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 section 14, Translation Limits.

stuff, nostuff

Format: stuff

nostuff [= <section type>[,…]]

<section type>: { B | D | C | W }

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).

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
	2	C_2
	1	C_1
Initialized variables	4	D
	2	D_2
	1	D_1
Uninitialized variables	4	B
	2	B_2
	1	B_1
switch statement branch table	4	W
	2	W_2
	1	W_1

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.

The default for these options is **stuff**.

Example:

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

<When **stuff** option is specified>

```
    .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,ALIGN=2
    .glb     _ST
_ST
    .blkb    2
```

<When **nostuff** option is specified>

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

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

instalign4, instalign8, noinstalign

Format: instalign4[={loop|inmostloop}]

instalign8[={loop|inmostloop}]

noinstalign

Description: These options align instructions at branch destinations.

When the **instalign4** and **instalign8** options are specified, the instruction at the location address is aligned to the 4-byte boundary and 8-byte boundary, respectively.

The default for these options is **noinstalign**.

Instruction alignment is performed only when the instruction at the specified location exceeds the address which is a multiple of the alignment value (4 or 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.
Each option has specifications targeting the following usages.

instalign8:	When attempting to improve the speed of CPUs with a 64-bit instruction queue (mainly RX600 Series)
instalign4:	When attempting to improve the speed of CPUs with a 32-bit instruction queue (mainly RX200 Series)

noinstalign: When not expecting the effect of this function or when emphasizing the code size

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.

*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.

Example: <C source file>

```
long 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
        ADD     #01H,R1
        RTS
        .END
```

2.3 List Options

Table 2.5 List Options

No.	Option	Dialog Menu	Description
1	listfile[=<file name>]	C/C++ <List>	Outputs a source list file.
	nolistfile	[Generate list file]	Does not output a source list file.
2	show = <sub>[,…]	C/C++ <List> [Contents :]	Specifies the contents of the source list file.
	<sub>:		
	{source		Outputs the C/C++ source file.
	| conditionals		Outputs the statements unsatisfied in conditional assembly.
	| definitions		Outputs the information before **.DEFINE** replacement.
	| expansions }		Outputs the assembler macro expansion statements.

listfile, nolistfile

Format: listfile[=<file name>]

nolistfile

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.

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

If <file name> is not specified, the source file name with the extension replaced with **lst** is used as the source list file name.

The default for these options is **nolistfile**.

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 section 7.2, Source List.

show

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 2.6 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 section 7.2, Source List.

2.4 Optimize Options

The options related to optimization may not be applied depending on the condition. Confirm through the output code whether the relevant optimization has been applied.

Table 2.7 Optimize Options

No.	Option	Dialog Menu	Description
1	optimize = { 0 | 1 | 2 | max }	C/C++ <Optimize> [Optimize level :]	Specifies the optimization level.
2	goptimize	C/C++ <Optimize> [Inter-module optimization]	Outputs additional information for inter-module optimization.
3	speed size	C/C++ <Optimize> [Speed or size :] [Optimize for speed :] [Optimize for size :]	Selects the optimization type. Optimizes with emphasis on execution performance. Optimizes with emphasis on code size.
4	loop[=<numeric value>]	C/C++ <Optimize> [Details] [Miscellaneous] [Loop expansion :]	Expands a loop under the condition of loop expansion maximum number = <numeric value>.
5	inline = <file name>[,…] noinline	C/C++ <Optimize> [Details] [Inline] [Automatic inline expansion :]	Performs inline expansion automatically. Does not perform inline expansion automatically.
6	file_inline = <file name>[,…]	C/C++ <Optimize> [Details] [Inline] [Inline file path]	Specifies a file for inter-file inline expansion.

No.	Option	Dialog Menu	Description
7	case = { ifthen	C/C++ <Optimize>	Expands by **if_then** method.
		[Details]	Expands by jumping to a table.
		[Miscellaneous]	
		[Switch statement :]	The compiler selects the expansion method.
8	volatile	C/C++ <Optimize>	Handles external variables as if they are **volatile** qualified.
		[Details]	
	novolatile	[Global variables]	Does not handle external variables as if they are **volatile** qualified.
		[Treat global variables as volatile qualified]	
9	const_copy	C/C++ <Optimize>	Enables constant propagation of **const** qualified external variables.
		[Details]	
	noconst_copy	[Global variables]	Disables constant propagation of **const** qualified external variables.
		[Propagate variables which are const qualified :]	
10	const_div	C/C++ <Optimize>	Performs constant division (residue) by an instruction sequence using multiplication.
		[Details]	
	noconst_div	[Miscellaneous]	Performs constant division (residue) by an instruction sequence using division.
		[Division by constant :]	
11	library = { function	C/C++ <Optimize>	Calls library functions.
		[Details]	Performs instruction expansion of several library functions.
		[Miscellaneous]	
		[Library function :]	
12	scope	C/C++ <Optimize>	Divides optimizing ranges.
	noscope	[Details]	Does not divide optimizing ranges.
		[Miscellaneous]	
		[Divide the optimization range :]	
13	schedule	C/C++ <Optimize>	Schedules instructions.
	noschedule	[Details]	Does not schedule instructions.
		[Miscellaneous]	
		[Schedule instructions :]	

No.	Option	Dialog Menu	Description
14	map=<file name>	C/C++ <Optimize> [Optimization for access to external variables :]	Optimizes accesses to external variables.
	smap	[Inter-module]	Optimizes accesses to external variables which are defined in the file to be compiled.
	nomap	[None]	Disables optimization for accesses to external variables.
15	approxdiv	C/C++ <Optimize> [Details] [Miscellaneous] [Approximate a floating-point constant division]	Converts floating-point constant division into multiplication.
16	enable_register	C/C++ <Optimize> [Details] [Miscellaneous] [Enable register declaration]	Allocates preferentially the variables with **register** storage class specification to registers.
17	simple_float_conv	C/C++ <Optimize> [Details] [Miscellaneous] [Not check the range in conversion between floating-point number and integer]	Omits part of the type conversion processing for the floating type.
18	fpu	C/C++ <Optimize> [Details] [Miscellaneous] [Use floating-point arithmetic instructions :]	Outputs an object that uses **FPU** instructions.
	nofpu		Outputs an object that does not use **FPU** instructions.
19	alias = { noansi	C/C++ <Optimize> [Details...] [Miscellaneous] [Optimization considering type of object indicated by pointer]	Does not perform optimization considering the type of the data indicated by the pointer.
	| ansi }		Performs optimization considering the type of the data indicated by the pointer.

No.	Option	Dialog Menu	Description
20	float_order	C/C++ <Other> [Miscellaneous options:] [Change operation order for floating-point expression aggressively]	Optimizes modification of the operation order in a floating-point expression.

optimize

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.

The default for this option is **optimize=2**.

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

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, size

Format: speed

size

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

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

Remarks: When the **speed** or **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.

Table 2.8 Specified Options

- When **optimize=max** is specified

	Loop Expansion	Inline Expansion	Converting Constant Division into Multiplication	Scheduling Instructions	Constant Propagation of const Qualified Variables	Dividing Optimizing Ranges	Optimizing External Variable Accesses	Optimization Considering the Type of the Data Indicated by the Pointer
speed	loop=8	inline=250	const_div	schedule	const_ copy	noscope	map* nomap*	alias=ansi
size	loop=1	inline=0	noconst_div	schedule	const_ copy	noscope	map* nomap*	alias=ansi

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**.

- When **optimize=2** is specified

	Loop Expansion	Inline Expansion	Converting Constant Division into Multiplication	Scheduling Instructions	Constant Propagation of const Qualified Variables	Dividing Optimizing Ranges	Optimizing External Variable Accesses	Optimization Considering the Type of the Data Indicated by the Pointer
speed	loop=2	inline=100	const_div	schedule	const_ copy	scope	nomap	alias=noansi
size	loop=1	noinline	noconst_div	schedule	const_ copy	scope	nomap	alias=noansi

- When **optimize=0** or **optimize=1** is specified

	Loop Expansion	Inline Expansion	Converting Constant Division into Multiplication	Scheduling Instructions	Constant Propagation of const Qualified Variables	Dividing Optimizing Ranges	Optimizing External Variable Accesses	Optimization Considering the Type of the Data Indicated by the Pointer
speed	loop=1	noinline	const_div	noschedule	noconst_ copy	scope	nomap	alias=noansi
size	loop=1	noinline	noconst_div	noschedule	noconst_ copy	scope	nomap	alias=noansi

loop

Format: loop[=<numeric value>]

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.

inline, noinline

Format: inline[=<numeric value>]

noinline

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).

When the **inline** option is specified with no numeric value, **inline=100** is assumed.

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

The default for these options is determined based on the **optimize** option and **speed** or **size** option specifications. For details, refer to the **speed** or **size** option.

Remarks: Inline expansion is attempted for all functions for which **#pragma inline** has been specified or with an **inline** specifier whether other options have been specified or not. 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.

file_inline

Format: file_inline=<file name>[,…]

Description: This option performs inline expansion for functions that extend across files for the files specified with <file name>.

Multiple files can be specified by separating them with a comma (,).

Example:

```
<a.c>
func(){
   g();
 }
<b.c>
   g(){
   h();
 }
```

By compiling a program with **ccrx. -inline -file_inline=b.c a.c** specified, calling of function **g()** in **a.c** is expanded as follows:

```
func(){
   h();
 }
```

Remarks: The **file_inline** option is valid only when the **inline** option or **#pragma inline** has been specified.

If an **extern** function is defined with the same name in more than one file specified with the **file_inline** option, correct operation is not guaranteed (a single function definition randomly selected is used for inline expansion).

The extension of the file name specified by <file name> cannot be omitted.

A file to be compiled cannot be specified with the **file_inline** option.

A wildcard (* or ?) cannot be specified for <file name>.

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

case

Format: case={ ifthen | table | 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.

The default for this option is **case=auto**.

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, novolatile

Format: volatile

novolatile

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.

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.

The default for these options is **novolatile**.

const_copy, noconst_copy

Format: const_copy

noconst_copy

Description: When **const_copy** is specified, constant propagation is performed even for **const** qualified global variables.

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

The default for these options is **const_copy** when the **optimize=2** or **optimize=max** option has been specified. For any other case, the default for these options is **noconst_copy**.

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

const_div, noconst_div

Format: const_div

noconst_div

Description: When **const_div** is specified, divisions and residues of integer constants in the source file are converted into instruction sequences using multiplications.

When **noconst_div** is specified, divisions and residues of integer constants in the source file are converted into instruction sequences using divisions.

The default for these options is **const_div** when the **speed** option has been specified and **noconst_div** when the **size** option has been specified.

Remarks: Constant multiplication that can be performed through shift operations and division and residue that can be performed through bitwise AND operations cannot be controlled by the **const_div** and **noconst_div** options.

library

Format: library = { function | intrinsic }

Description: When **library=function** is specified, all library functions are called.

When **library=intrinsic** is specified, instruction expansion is performed for **abs()**, **fabsf()**, and library functions which can use string manipulation instructions.

The default for this option is **library=intrinsic**.

scope, noscope

Format: scope

noscope

Description: When the **scope** option is specified, the optimizing ranges of the large-size function are divided into many sections before compilation.

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.

The default for these options is **noscope** when the **optimize=max** option has been specified. For any other case, the default for these options is **scope**.

schedule, noschedule

Format: schedule

noschedule

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

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.

The default for these options is **schedule** when the **optimize=2** or **optimize=max** option has been specified. For any other case, the default for these options is **noschedule**.

map, smap, nomap

Format: map[= <file name>]

smap

nomap

Description: These options optimize 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 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.

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** or **output=mot** is specified]
Specify only **map** (not necessary when **optimize=max** is specified). Compilation and linkage are automatically performed twice, and a code in which the base address is set based on external symbol allocation information is generated.

[**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**.

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

The default for these options is **map** when the **optimize=max** option has been specified. For any other case, the default for these options is **nomap**.

Example: <C source file>

```
long A,B,C;
```

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

<Output code>
(1) When optimization is not performed

```
_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]
```

(2) When optimization is performed

```
_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.
```

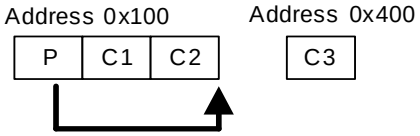
Remarks: When the order of the definitions of global variables or static variables has been changed, a new external symbol-allocation information file must be created. If any option other than the **map** option in the previous compilation differs from the one in the current compilation, or if any contents 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.

approxdiv

Format: approxdiv

Description: This option converts divisions of floating-point constants into multiplications of the corresponding reciprocals as constants.

To be specific, when there is an expression of (variable ÷ divisor) with the divisor being a constant, a code with the expression converted into (variable × reciprocal of divisor) will be generated.

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

enable_register

Format: enable_register

Description: This option allocates preferentially the variables with **register** storage class specification to registers.

Remarks: When the **message** option is specified, if a variable cannot be allocated to a register, the following information message will be output:

```
C0102 (I) Register is not allocated to "variable name" in
"function name"
```

Note however that this message will not be output if a parameter is not allocated to a register.

simple_float_conv

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.

(1) Type conversion from 32-bit floating type to unsigned integer type
(2) Type conversion from unsigned integer type to 32-bit floating type
(3) Type conversion from integer type to 64-bit floating type via 32-bit floating type

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.

Example 1: <Type conversion from 32-bit floating type to unsigned integer type>

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

When this option is not specified:

```
_func1:
    MOV.L   R1,R5
    FCMP    #4F000000H,R1
    BLT     L12
    FADD    #0CF800000H,R5
L12:
    FTOI    R5,R1
    RTS
```

When this option is specified:

```
        FTOI    R1,R1
        RTS
```

Example 2: <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,R4
    AND     #1,R1
    OR      R4,R1
    ITOF    R1,R4
    FMUL    #40000000H,R4
    BRA     L16
L15:
    ITOF    R1,R4
L16:
    MOV.L   R4,R1
    RTS
```

When this option is specified:

```
        ITOF    R1,R1
        RTS
```

Example 3: <Type conversion from integer type to 64-bit floating type via 32-bit floating type>

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

```
double func3(long l)
{
    return (double)(float)(double)l;
}
```

When this option is not specified:

```
_func3:
    BSR     __COM_CONV32sd
    BSR     __COM_CONVdf
    BRA     __COM_CONVfd
```

When this option is specified:

```
BRA     __COM_CONV32sd
```

fpu, nofpu

Format: fpu

nofpu

Description: These options select whether to use **FPU** instructions when generating a code to perform floating-point operations.

When the **fpu** option is specified, a code using **FPU** instructions is generated.

When the **nofpu** option is specified, a code using runtime library calls instead of **FPU** instructions is generated.

The default for these options is **fpu** when **RX600** is selected* as the CPU and **nofpu** when **RX200** is selected* as the CPU.

Note: * This means to be selected by either the **cpu** option or the environment variable **CPU_RX**.

Remarks: For details of the **FPU** instructions, refer to the RX Family Software Manual.

When **RX200** is selected as the CPU, an error will occur if **fpu** is specified.

If **-fpu** is enabled, when an operation is performed to a denormalized number so that the result is the same as the previous one (e.g., 1 is multiplied), the operation result may turn into 0.

Example: If **-fpu -denormalize=on** is enabled, the execution result of the following source code becomes **f = 0.0** or **f = 1E-40**.

```
float func(void)
{
  float f = 1E-40;
  f*=1;
  return f;
}
```

alias

Format: alias = { noansi | ansi }

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.

The default for this option is **alias=noansi**.

Example:

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

[When **alias=noansi** is specified]

Note: 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]

Note: 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

Format: float_order

Description: This option optimizes modification of the operation order in a floating-point expression.

Specifying the **float_order** option generally improves the object performance compared to when not specifying it. However, the accuracy of operations may differ from that when **float_order** is not specified.

Example:
```
float a,b,c;
f()
{
    a = b * 100.0f + c * 100.0f;
}
```

If the **float_order** option is enabled, the floating-point expression in this example is replaced with an operation expression equivalent to **a = (b + c) * 100.0f**.

Remarks: This option is enabled only when **optimize=2** or **optimize=max** is specified.

This option is ignored when **optimize=0** or **optimize=1** is specified. In such a case, warning C1301(W) is displayed.

2.5 Microcontroller Options

Table 2.9 Microcontroller Options

No.	Option	Dialog Menu	Description
1	cpu = { rx600	CPU [CPU :]	Generates an instruction code for the RX600 Series.
	| rx200 }		Generates an instruction code for the RX200 Series.
2	endian = {	CPU [Endian :]	Specifies the endian type for data.
	big		Big endian
	| little }		Little endian
3	round = { zero	CPU	Rounds to zero.
	| nearest }	[Details] [Detail] [Round to :]	Rounds to nearest.
4	denormalize = { off	CPU [Details]	Handles denormalized numbers as zeros.
	| on }	[Detail] [Denormalized number allower as a result :]	Handles denormalized numbers as they are.
5	dbl_size = { 4	CPU [Details] [Detail] [Precision of double :] [Single precision] [Double precision]	Handles the **double** type and **long double** type in single precision.
	| 8 }		Handles the **double** type and **long double** type in double precision.
6	int_to_short	CPU [Details] [Detail] [Replace from int with short]	Replaces the **int** type with the **short** type and the **unsigned int** type with the **unsigned short** type.

No.	Option	Dialog Menu	Description
7	signed_char	CPU [Details] [Detail] [Sign of char :]	Handles the **char** type as **signed char**.
	unsigned_char		Handles the **char** type as **unsigned char**.
8	signed_bitfield	CPU [Details] [Detail] [Sign of bit field :]	The sign of a bit-field is interpreted as **signed**.
	unsigned_bitfield		The sign of a bit-field is interpreted as **unsigned**.
9	auto_enum	CPU [Details] [Detail] [enum size is made the smallest]	Automatically selects the enumeration type size.
10	bit_order = { left |	CPU [Details] [Detail] [Bit field order :] [Upper_bit] [Lower_bit]	Stores bit-field members from the left.
	right }		Stores bit-field members from the right.
11	pack	CPU [Details] [Detail] [Pack struct, union and class]	Assumes the boundary alignment value for structure members is 1.
	unpack		Follows the boundary alignment.
12	exception	CPU [Details] [Detail] [Use try, throw and catch of C++]	Enables the exception handling function.
	noexception		Disables the exception handling function.

No.	Option	Dialog Menu	Description
13	rtti= { on | off}	CPU [Details] [Detail] [Use dynamic_cast and typeid of C++]	Enables **dynamic_cast** and **typeid**. Disables **dynamic_cast** and **typeid**.
14	fint_register = { 0 | 1 | 2 | 3 | 4 }	CPU [Fast interrupt vector register :] [None] [R13] [R12,R13] [R11,R12,R13] [R10,R11,R12,R13]	Specifies the general registers used only in fast interrupt functions. No registers are used only for fast interrupts. R13 is used only for fast interrupts. R13 and R12 are used only for fast interrupts. R13 to R11 are used only for fast interrupts. R13 to R10 are used only for fast interrupts.
15	branch = { 16 | 24 | 32 }	CPU [Details] [Detail] [Width of divergence of function :]	Guarantees the branch width size is within 16 bits. Guarantees the branch width size is within 24 bits. Does not limit the branch width size.
16	base = { rom = <register> | ram= <register> | <address value> = <register> }	CPU [Base register :]	Specifies the base register for ROM. Specifies the base register for RAM. Specifies the base register that sets the address value.
17	patch = { rx610 }	CPU [Changes code generation :]	Avoids a problem specific to the CPU type. The MVTIPL instruction should not be generated (for RX610 Group).

No.	Option	Dialog Menu	Description
18	pic	CPU [Details...] [PIC/PID] [Generate the code section as a position-independent code]	Enables the PIC function.
19	pid = { 16 | 32 }	CPU [Details...] [PIC/PID] [Generate the data section as a position-independent data] [Offset width:] [16bits] [32bits]	Enables the PID function. 16-bit (64 Kbytes to 256 Kbytes) addressing mode is supported 32-bit (4 Gbytes) addressing mode is supported
20	nouse_pid_register	CPU [Details...] [PIC/PID] [No use the register for PID]	Does not use the PID register for code generation.
21	save_acc	CPU [Details…] [Detail] [The saved and restored code of the accumulator]	Saves and restores **ACC** using the interrupt function.

cpu

Format: cpu={ rx600 | rx200 }

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: 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.

endian

Format: endian={ big | 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 endian type can also be specified by the **#pragma endian** extension. If both this option and a **#pragma** extension are specified, the **#pragma** specification takes priority.

The default for this option is **endian=little**.

round

Format: round={ zero | 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.

The default for this option is **round=nearest**.

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

Format: denormalize={ off | on }

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.

The default for this option is **denormalize=off**.

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

Format: dbl_size={ 4 | 8 }

Description: This option specifies the precision of the **double** type and **long double** type.

When **dbl_size=4** is specified, the **double** type and **long double** type are handled as the single-precision floating type (4 bytes).

When **dbl_size=8** is specified, the **double** type and **long double** type are handled as the double-precision floating type (8 bytes).

The default for this option is **dbl_size=4**.

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

Format: int_to_short

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 C1804(W) 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 C1804(W) 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.

signed_char, unsigned_char

Format: signed_char

unsigned_char

Description: These options specify the sign of the **char** type with no sign specification.

When **signed_char** is specified, the value is handled as the **signed char** type.

When **unsigned_char** is specified, the value is handled as the **unsigned char** type.

The default for these options is **unsigned_char**.

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, unsigned_bitfield

Format: signed_bitfield

unsigned_bitfield

Description: These options specify the sign of the bit-field type with no sign specification.

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

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

The default for these options is **unsigned_bitfield**.

auto_enum

Format: auto_enum

Description: This option processes the enumerated data qualified by **enum** as the minimum data type with which the enumeration value can fit in.

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

The possible enumeration values correspond to the data types as shown in the following table.

Table 2.10 Correspondences between Possible Enumeration Values and Data Types

Enumerator		
Minimum Value	**Maximum Value**	**Data Type**
–128	127	signed char
0	255	unsigned char
–32768	32767	signed short
0	65535	unsigned short
Other than above	Other than above	signed long

bit_order

Format: bit_order = { left | 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.

The default for this option is **bit_order=right**.

pack, unpack

Format: pack

unpack

Description: These options specify the boundary alignment value for structure members and class members.

The boundary alignment value for structure members can also be specified by the **#pragma pack** extension. If both this option and a **#pragma** extension are specified, the **#pragma** specification takes priority. The boundary alignment value for structures and classes equals the maximum boundary alignment value for members.

The default for these options is **unpack**.

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

Table 2.11 Boundary Alignment Values for Structure Members and Class Members When pack/unpack Option is Specified

Member Type	pack	unpack	Not Specified
(signed) char	1	1	1
(unsigned) short	1	2	2
(unsigned) int*, (unsigned) long, (unsigned) long long, floating type, and pointer type	1	4	4

Note: * Becomes the same as **short** when the **int_to_short** option is specified.

exception, noexception

Format: exception

noexception

Description: When the **exception** option is specified, the C++ exceptional handling function (**try**, **catch**, **throw**) is enabled.

When the **noexception** option is specified, the C++ exceptional handling function (**try**, **catch**, **throw**) is disabled.

The default for these options is **noexception**.

Remarks: In order to use the C++ exceptional handling function among files, perform the following:
(1) Specify **rtti=on**.
(2) Do not specify the **noprelink** option in the optimizing linkage editor.

The **exception** option can be specified only at C++ compilation. The **exception** 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

Format: rtti={ on | 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.

The default for this option is **rtti=off**.

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** 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.

fint_register

Format: fint_register = {0 | 1 | 2 | 3 | 4 }

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 2.12 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

The default for this option is **fint_register=0**.

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

Format: branch = { 16 | 24 | 32 }

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.

The default for this option is **branch=24**.

base

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 all performed relative to the specified register A. Note that the total size of the constant area section must be within 64 Kbytes to 256 Kbytes*.

When **base=ram=<register B>** is specified, accesses to initialized variables and uninitialized variables are all performed relative to the specified register B. Note that the total RAM data size must be within 64 Kbytes to 256 Kbytes*.

When **<address value>=<register C>** is specified, accesses to an area within 64 Kbytes to 256 Kbytes* from the address value are performed relative to the specified register C.

Note: * 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 C1801(W) is output as a warning and the selection of **base=rom=<register>** is disabled.

patch

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

Format: pic

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.

Examples: <Example 1> 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
```

<Example 2> 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 C1801(W) 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 C6698(E) 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 section 8.4.7, Application Startup, instead of section 8.3, Startup.

For the PIC function, also refer to section 8.4, Usage of PIC/PID Function.

pid

Format: pid[={ 16 | 32 }]

Description: The constant area sections **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 2.13 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 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: <Example 1> 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
```

<Example 2> 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 C6699(E) 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 section 8.4.7, Application Startup, instead of section 8.3, 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 C1805(W) or C1806(W) 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. C1805(W) is output when an externally referenced variable that is **const** qualified is not in the constant area. C1806(W) is output when an externally referenced variable that is not **const** qualified is in the constant area. In either case, the displayed variable is handled as PID.

In C++ or EC++ compilation, the **pid** option cannot be selected. If selected, message C1801(W) 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 C1801(W) 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, error C2028(E) will occur.

If the **pid** option and **nouse_pid_register** option are selected simultaneously, error C3305(F) will occur.

For details of the application and PID function, refer to section 8.4, Usage of PIC/PID Function.

nouse_pid_register

Format: nouse_pid_register

Description: This option generates code without using the PID register. For more on the PID register, refer to the description of 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.

Note that selecting this option does not enable the PID function.

Remarks: If the **nouse_pid_register** option and **pid** option are selected simultaneously, error C3305(F) will occur.

If a register specified by the **nouse_pid_register** option is also specified by the **base** option, error C2028(E) will occur.

For details of the master program, application program, and PID function, refer to section 8.4, Usage of PIC/PID Function.

save_acc

Format: save_acc

Description: This option generates the saved and restored code of the accumulator (**ACC**) for interrupt functions.

Remarks: The generated saved and restored code is the same code generated when **acc** is selected in **#pragma interrupt**. For the actual saved and restored code, refer to the description of **acc** and **no_acc** in **#pragma interrupt** of section 9.2.1, #pragma Extension Specifiers and Keywords.

2.6 Assemble and Linkage Options

Table 2.14 Assemble and Linkage Options

No.	Option	Dialog Menu	Description
1	asmcmd=<file name>	—	Specifies the **asrx** options with a subcommand file.
2	lnkcmd=<file name>	—	Specifies the **optlnk** options with a subcommand file.
3	asmopt=['']<assembler option> [Δ<assembler option>…][']	—	Specifies the **asrx** options.
4	lnkopt=['']<linkage option> [Δ<linkage option>…][']	—	Specifies the **optlnk** options.

asmcmd

Format: asmcmd=<file name>

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

Example:
```
ccrx -cpu=rx600 -asmcmd=file.sub sample.c
```
The above description has the same meaning as the following two command lines:
```
ccrx -cpu=rx600 -output=src sample.c
asrx -cpu=rx600 -subcommand=file.sub sample.src
```

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

lnkcmd

Format: lnkcmd=<file name>

Description: This option specifies the linkage options to pass to **optlnk** with a subcommand file.

Example:
```
ccrx -cpu=rx600 -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 -cpu=rx600 -output=src tp1.c tp2.c
asrx -cpu=rx600 tp1.src tp2.src
optlnk –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.

asmopt

Format: asmopt=["]<assembler option>[Δ<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 –cpu=rx600 –asmopt="–chkpm" sample.c
```
The above description has the same meaning as the following two command lines:
```
ccrx –cpu=rx600 –output=src sample.c
asrx –cpu=rx600 –chkpm sample.src
```

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

lnkopt

Format: lnkopt=["]<linkage option>[Δ<linkage option>…]["]

Description: This option specifies the linkage options to pass to **optlnk** with a string.

Multiple options can be specified by enclosing them with double-quote marks (").

Example:
```
ccrx -cpu=rx600 -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 -cpu=rx600 -output=src tp1.c tp2.c
asrx -cpu=rx600 tp1.src tp2.src
optlnk -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.

2.7 Other Options

Table 2.15 Other Options

No.	Option	Dialog Menu	Description
1	logo	—	Outputs the copyright.
	nologo	(**nologo** is always valid)	Disables output of the copyright.
2		C/C++ <Source> [Show entries for:] [Source file] [Input character code:]	Specifies the character code of an input program.
	euc		EUC code
	sjis		SJIS code
	latin1		ISO-Latin1 code
	utf8		UTF-8 code
3		C/C++ <Object> [Output character code:]	Specifies the character code of an output assembly-language file.
	outcode = { euc		EUC code
	| sjis		SJIS code
	| utf8 }		UTF-8 code
4	subcommand = <file name>	—	Includes command options from a file specified by <file name>.

logo, nologo

Format: logo

nologo

Description: These options control the copyright output.

When the **logo** option is specified, the copyright notice is output.

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

The default for these options is **logo**.

euc, sjis, latin1, utf8

Format: euc

sjis

latin1

utf8

Description: Each option specifies the character code to handle the characters in strings, character constants, and comments.

The options correspond to the character codes as shown in the following table.

The default for these options is **sjis**.

Table 2.16 Correspondences between Options and Character Codes (euc, sjis, latin1, utf8)

Option	Character Code
euc	EUC code
sjis	SJIS code
latin1	ISO-Latin1 code
utf8	UTF-8 code

Remarks: The **utf8** option is valid only when the **lang=c99** option has been specified.

outcode

Format: outcode = { euc | sjis | utf8 }

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.

The default for this option is **outcode=sjis**.

Table 2.17 Correspondences between Options and Character Codes (outcode)

Option	Character Code
euc	EUC code
sjis	SJIS code
utf8	UTF-8 code

Remarks: The **utf8** option is valid only when the **lang=c99** option has been specified.

subcommand

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.

Section 3 Library Generator Options

3.1 Library Generator Options

Table 3.1 Library Generator Options

No.	Option	Dialog Menu	Description
1	head=<sub>[,…] <sub>:{ all | runtime | ctype | math | mathf | stdarg | stdio | stdlib | string | ios | new | complex | cppstring | c99_complex | fenv | inttypes | wchar | wctype }	Standard Library <Standard Library> [Category:]	Specifies a configuration library. All library functions and runtime library Runtime library ctype.h (C89/C99) and runtime library math.h (C89/C99) and runtime library mathf.h (C89/C99) and runtime library stdarg.h (C89/C99) and runtime library stdio.h (C89/C99) and runtime library stdlib.h (C89/C99) and runtime library string.h (C89/C99) and runtime library ios (EC++) and runtime library new (EC++) and runtime library complex (EC++) and runtime library string (EC++) and runtime library complex.h (C99) and runtime library fenv.h (C99) and runtime library inttypes.h (C99) and runtime library wchar.h (C99) and runtime library wctype.h (C99) and runtime library
2	output = <file name>	Standard Library <Object> [Output file path:]	Specifies an output library file name.
3	nofloat	Standard Library <Object> [I/O functions:] [The functional cutdown version 1]	Creates a simple I/O function.

No.	Option	Dialog Menu	Description
4	reent	Standard Library <Object> [Generate reentrant library]	Creates a reentrant library.
5	lang = { c | c99 }	Standard Library <Standard Library> [Library configuration:] [C(C89)] [C99]	Selects the set of functions available from the C standard library.
6	simple_stdio	Standard Library <Object> [I/O functions:] [The functional cutdown version 2]	Creates a functionally cut down version of the set of I/O functions.
7	logo nologo	- (nologo is always valid)	Outputs the copyright. Disables output of the copyright.

head

Format: head=<sub>[,…]

<sub>:{ all
| runtime | ctype | math | mathf | stdarg | stdio | stdlib | string | ios | new
| complex | cppstring | c99_complex | fenv | inttypes | wchar | wctype
}

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.

The default for this option is **head=all**.

output

Format: output=<file name>

Description: This option specifies an output file name.

The default for this option is **output=stdlib.lib**.

nofloat

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

Format: reent

Description: This option creates reentrant libraries. Note that the **rand** and **srand** functions are not reentrant libraries.

Remarks: 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

Format: lang = { c | c99 }

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.

The default for this option is **lang=c**.

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

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.

logo, nologo

Format: logo

nologo

Description: These options control the copyright output.

When the **logo** option is specified, the copyright notice is output.

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

The default for these options is **logo**.

3.2 Compiler Options that Become Invalid

In addition to the options in section 3.1, 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 3.2 Invalid Options

No.	Options that Become Invalid	Conditions for Invalidation	Option Selected at Library Configuration When Made Invalid
1	lang	Always invalid	None
2	include	Always invalid	None
3	define	Always invalid	None
4	undefined	Always invalid	None
5	message nomessage	Always invalid	nomessage
6	change_message	Always invalid	None
7	file_inline_path	Always invalid	None
8	comment	Always invalid	None
9	check	Always invalid	None
10	output	Always invalid	output=obj
11	noline	Always invalid	None
12	debug nodebug	Always invalid	nodebug
13	object noobject	Always invalid	None
14	listfile nolistfile show	Always invalid	nolistfile
15	file_inline	Always invalid	None
16	asmcmd	Always invalid	None
17	lnkcmd	Always invalid	None
18	asmopt	Always invalid	None

No.	Options that Become Invalid	Conditions for Invalidation	Option Selected at Library Configuration When Made Invalid
19	lnkopt	Always invalid	None
20	logo nologo	Always invalid	nologo
21	euc sjis latin1 utf8	Always invalid	None
22	outcode	Always invalid	None
23	subcommand	Always invalid	None
24	alias	Always invalid	alias=noansi
25	pic pid	**lang=cpp** or at C++ source compilation*	None

Note: * Warning C1801(W) is output.

Section 4 Assembler Options

4.1 Source Options

Table 4.1 Source Options

No.	Option	Dialog Menu	Description
1	include=<path name>[,...]	Assembly <Source> [Show entries for:] [Include file directories]	Specifies the name of the path to the folder that stores the include file.
2	define=<sub>[,...] <sub>: <replacing symbol name>=<string>	Assembly <Source> [Show entries for:] [Defines]	Defines <string> as <replacing symbol name>.
3	chkpm	Assembly <Other> [Miscellaneous options:]	Checks for a privileged instruction.
4	chkfpu	Assembly <Other> [Miscellaneous options:]	Checks for a floating-point operation instruction.
5	chkdsp	Assembly <Other> [Miscellaneous options:]	Checks for a DSP instruction.

include

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 (,).

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:
```
asrx -include=c:\usr\inc,c:\usr\rxc test.src
```
Folders **c:\usr\inc** and **c:\usr\rxc** are searched for the include file.

define

Format: define=<sub>[,...]

<sub>: <replacing symbol name> = <string>

Description: This option replaces the replacing symbol 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

Format: chkpm

Description: This option outputs warning A1011 when a privileged instruction is used in the source file.

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

chkfpu

Format: chkfpu

Description: This option outputs warning A1012 when a floating-point operation instruction is used in the source file.

Remarks: For details of the floating-point operation instructions, refer to the RX Family Software Manual.

chkdsp

Format: chkdsp

Description: This option outputs warning A1013 when a DSP instruction is used in the source file.

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

4.2 Object Options

Table 4.2 Object Options

No.	Option	Dialog Menu	Description
1	output= <output file name>	Assembly <Object> [Output directory]	Specifies the relocatable file name.
2	debug nodebug	Assembly <Object> [Generate debug information]	Outputs debugging information. Does not output debugging information.
3	goptimize	Assembly <Object> [Inter-module optimization]	Outputs additional information for inter-module optimization.

output

Format: output=<output file name>

Description: This option specifies the name of the relocatable file to be output.

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**.

If this option is not specified, the source file name with the extension replaced with **.obj** is used as the output relocatable file name.

debug, nodebug

Format: debug

nodebug

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

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

The default for this option is **nodebug**.

goptimize

Format: goptimize

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.

4.3 List Options

Table 4.3 List Options

No.	Option	Dialog Menu	Description
1	listfile[=<file name>]	Assembly <List>	Outputs a source list file.
	nolistfile	[Generate list file]	Does not output a source list file.
2	show = <sub>[,...]	Assembly <List> [Generate list file]	Specifies the contents of the output source list file.
	<sub>: { conditionals	[Source program:]	Outputs the statements unsatisfied in conditional assembly.
	| definitions		Outputs the information before replacement specified with **.DEFINE**.
	| expansions }		Outputs the macro expansion statements.

listfile, nolistfile

Format: listfile[=<file name>]

nolistfile

Description: These options specify whether to output a source list file.

When the **listfile** option is specified, a source list file is output. The name of the file can also be specified.

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

<file name> should be specified according to the rules described in section 7.1, Naming Files.

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.

The default for this option is **nolistfile**.

show

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 4.4 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.

4.4 Microcontroller Options

Table 4.5 Microcontroller Options

No.	Option	Dialog Menu	Description
1	cpu = { rx600	CPU [CPU:]	Generates a relocatable file for the RX600 Series.
	| rx200 }		Generates a relocatable file for the RX200 Series.
2	endian = { big	CPU [Endian:]	Big endian
	| little }		Little endian
3	fint_register = {	CPU [Fast interrupt register:]	Specifies general registers to be used only for fast interrupts.
	0		No registers are dedicated to fast interrupts
	| 1		R13 is dedicated to fast interrupts
	| 2		R13 and R12 are dedicated to fast interrupts
	| 3		R13 to R11 are dedicated to fast interrupts
	| 4 }		R13 to R10 are dedicated to fast interrupts
4	base = <sub>[,...]	CPU	
	<sub>: { rom = <register>	[Base register:]	Specifies the base register for ROM.
	| ram= <register>		Specifies the base register for RAM.
	| <address> = <register>}		Specifies the base register for SFR.
5	patch = {	CPU [Changes code generation for the CPU types :]	Avoids a problem specific to the CPU type.
	rx610 }		The **MVTIPL** instruction should not be used (for RX610 Group).
6	pic	CPU [Details...] [PIC/PID] [Generate the code section as a position-independent code]	Generates an object with the PIC function enabled.

No.	Option	Dialog Menu	Description
7	pid = {	CPU [Details...]	Generates an object with the PID function enabled and selects the offset width.
	16	[PIC/PID]	16-bit (64 Kbytes to 256 Kbytes) addressing mode is supported.
	| 32 }	[Generate the data section as a position-independent data] [Offset width:] [16bits] [32bits]	32-bit (4 Gbytes) addressing mode is supported.
8	nouse_pid_register	CPU [Details...] [PIC/PID] [No use the register for PID]	Does not use the PID register for code generation.

cpu

Format: cpu={ rx600 | rx200 }

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: Suboptions will be added depending on the microcontroller products developed in the future.

When **rx200** is specified, writing floating-point operation instructions (**FADD**, **FCMP**, **FDIV**, **FMUL**, **FSUB**, **FTOI**, **ITOF**, and **ROUND**) which are not supported by the RX200 Series or writing **FPSW** in control registers for the **MVTC**, **MVFC**, **PUSHC**, and **POPC** instructions will cause an error.

endian

Format: endian={ big | 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 default for this option is **endian=little**.

fint_register

Format: fint_register = {0 | 1 | 2 | 3 | 4 }

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

Format: base = <sub>[,...]

<sub>: { 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 in assembly-language files. 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

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 A2113(E) will be output.

pic

Format: pic

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 section 8.4, Usage of PIC/PID Function.

pid

Format: pid[={ 16 | 32 }]

Description: This option generates a relocatable object that 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 4.6 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 A3111(F) will be output.

If the **pid** option and **nouse_pid_register** option are selected simultaneously, error A3103(F) will be output.

For the PID function, also refer to section 8.4, Usage of PIC/PID Function.

nouse_pid_register

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 A2058(E) will be output.

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 A3103(F) will be output.

If a register specified by the **nouse_pid_register** option is also specified by the **base** option, error A3112(F) will be output.

For the PID function, also refer to section 8.4, Usage of PIC/PID Function.

4.5 Other Options

Table 4.7 Other Options

No.	Option	Dialog Menu	Description
1	logo	-	Outputs the copyright.
	nologo	(nologo is always valid)	Disables output of the copyright.
2	subcommand = <file name>	-	Inputs command line specifications from a file.
3	euc	-	Selects EUC code.
	sjis		Selects SJIS code.
	latin1		Selects ISO-Latin1 code.

logo, nologo

Format: logo

nologo

Description: These options control the copyright output.

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

The default for this option is **logo**.

subcommand

Format: 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, sjis, latin1

Format: euc

sjis

latin1

Description: Each option specifies the character code to handle the characters in strings, character constants, and comments.

The options correspond to the character codes as shown in the following table.

Table 4.8 Correspondences between Options and Character Codes (euc, sjis, latin1)

Option	Character Code
euc	EUC code
sjis	SJIS code
latin1	ISO-Latin1 code

Section 5 Optimizing Linkage Editor Options

5.1 Option Specifications

5.1.1 Command Line Format

The format of the command line is as follows:

```
optlnk[{Δ<file name>|Δ<option string>}...]
    <option string>:-<option>[=<suboption>[,...]]
```

5.1.2 Subcommand File Format

The format of the subcommand file is as follows:

```
<option>{=|Δ}[<suboption>[,...]][Δ&][;<comment>]
&: means line continuous.
```

For details, refer to section 5.2.8, Subcommand File Option.

5.2 List of Options

In the command line format in the following sections, uppercase letters indicate abbreviations. Underlined characters indicate the default settings.

The format of the corresponding dialog menus in the High-performance Embedded Workshop is as follows:

Tab name <Category>[Item]....

The order of option description corresponds to that of the tabs and the categories in the High-performance Embedded Workshop.

The file name and path name should not include a parenthesis ("(" or ")").

5.2.1 Input Options

Table 5.1 Input Category Options

Item	Command Line Format	Dialog Menu	Specification
Input file	Input = <sub>[{,|Δ}...] <sub>: <file name> [(<module name>[,…])]	Link/Library <Input> [Show entries for :] [Relocatable files and object files]	Specifies input file. (Input file is specified without **input** on the command line.)
Library file	LIBrary = <file name>[,...]	Link/Library <Input> [Show entries for :] [Library files]	Specifies input library file.
Binary file	Binary = <sub> [,...] <sub>: <file name>(<section name> [:<boundary alignment>] [/<section attribute>] [,<symbol name>])	Link/Library <Input> [Show entries for :] [Binary files]	Specifies input binary file.
Symbol definition	DEFine = <sub>[,…] <sub>: <symbol name> = {<symbol name> |<numerical value>}	Link/Library <Input> [Show entries for :] [Defines:]	Defines undefined symbols forcedly. Defined as the same value of symbol name. Defined as a numerical value.
Execution start address	ENTry = { <symbol name>| <address>}	Link/Library <Input> [Use entry point :]	Specifies an entry symbol. Specifies an entry address.
Prelinker	NOPRElink	Link/Library <Input> [Prelinker control :]	Disables prelinker initiation.

Input Input File

Link/Library <Input>[Show entries for :][Relocatable files and object files]

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 Library File

Link/Library <Input>[Show entries for :][Library files]

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 — Binary File

Link/Library <Input>[Show entries for :][Binary files]

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 — Symbol Definition

Link/Library <Input>[Show entries for :][Defines]

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 — Execution Start Address

Link/Library <Input>[Use entry point :]

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.

NOPRElink Prelinker

Link/Library <Input>[Show entries for :][Prelinker control :]

Format: NOPRElink

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**.

5.2.2 Output Options

Table 5.2 Output Category Options

Item	Command Line Format	Dialog Menu	Specification
Output format	FOrm ={ Absolute | Relocate | Object | Library [= {S|U}] | Hexadecimal | Stype | Binary }	Link/Library <Output> [Type of output file :]	Absolute format Relocatable format Object format Library format HEX format S-type format Binary format
Debugging information	DEBug SDebug NODEBug	Link/Library <Output> [Debug information :]	Output (in output file) Debugging information file output Not output
Record size unification	REcord={ H16 | H20 | H32 | S1 | S2 | S3 }	Link/Library <Output> [Data record header :]	HEX record Expansion HEX record 32-bit HEX record S1 record S2 record S3 record
ROM support function	ROm = <sub>[,…] <sub>:<ROM section name> =<RAM section name>	Link/Library <Output> [Show entries for :] [ROM to RAM mapped sections:]	Reserves an area in RAM for the relocation of a symbol with an address in RAM.
Output file	OUtput = <sub>[,…] <sub>:<file name> [=<output range>] <output range>: {<start address> -<end address> |<section name>[:…]}	Link/Library <Output> [Show entries for :] [Output file path/ Messages] or [Divided output files:]	Specifies output file (range specification and divided output are enabled)
External symbol-allocation information file	MAp [= <file name>]	Link/Library <Output> [Generate external symbol-allocation information file]	Specifies output of the external symbol-allocation information file (for SuperH Family and RX Family)

Item	Command Line Format	Dialog Menu	Specification
Output to unused area	SPace [= {<numerical value> | Random}]	Link/Library <Output> [Specify value filled in unused area] [Output padding data]	Specifies a value to output to unused area
Information message	Message NOMessage [= <sub>[,…]] <sub>:<error code> [-<error code>]	Link/Library <Output> [Show entries for :] [Output file path/ Messages] [Repressed information level messages:]	Output No output (error number specification and range specification are enabled)
Notification of unreferenced defined symbol	MSg_unused	Link/Library <Output> [Show entries for :] [Notify unused symbol:]	Notifies the user of the defined symbol which is never referenced
Reduce empty areas of boundary alignment	DAta_stuff	Link/Library <Output> [Show entries for :] [Reduce empty areas of boundary alignment:]	Reduces empty areas generated as the boundary alignment of sections after compilation (for SuperH Family and H8, H8S, H8SX Family)
Specification of data record byte count	BYte_count=<numerical value>	Link/Library <Output> [Length of data record :]	Specifies the maximum byte count of a data record
CRC	CRc = <suboption> <suboption>: <address where the result is output>=<target range> [/<polynomial expression>] [:<endian>] <address where the result is output>: <address> <target range>: <start address>-<end address>[,...] <polynomial expression>: { CCITT | 16 } <endian>: {BIG | LITTLE}	Link/Library <Output> [Show entries for :] [Generate CRC code]	Calculates the cyclic redundancy check (CRC) value for the target range at linkage and outputs the result to the specified address.
Filling padding data at section end	PADDING	Link/Library <Output> [Padding]	Outputs padding data to the end of a section to make the section match the boundary alignment.
Address setting for specified vector number	VECTN=<suboption>[,...] <suboption>: <vector number>={<symbol> | | <address>}	Link/Library <Output> [Show entries for :] [Vector] [Specific vector :]	Assigns an address to the specified vector number in the variable vector table (for RX Family and M16C Series).

Item	Command Line Format	Dialog Menu	Specification
Address setting for unused variable vector area	VECT={<symbol>|<address>}	Link/Library <Output> [Show entries for :] [Vector] [Empty vector :]	Assigns an address to an unused area in the variable vector table (for RX Family and M16C Series).
utl30 information output	UTL	Link/Library <Output> [UTL information]	Outputs information for **UTL30** (for M16C Series)
Jump table output	JUMP_ENTRIES_FOR_PIC =<section name>[…]	Link/Library <Output> [Jump table output]	Outputs a jump table (for the PIC function of RX Family)

FOrm — Output Format

Link/Library <Output>[Type of output file :]

Format: FOrm = {Absolute | Relocate | Object | Library[={S | U}]}
| Hexadecimal | Stype | Binary}

Description: Specifies the output format.

When this option is omitted, the default is **form=absolute**. Table 5.3 lists the suboptions.

Table 5.3 Suboptions of Form Option

Suboption	Description
absolute	Outputs an absolute file
relocate	Outputs a relocatable file
object	Outputs an object file. This is specified when a module is extracted as an object file from a library with the **extract** option.
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**.
hexadecimal	Outputs a **HEX** file. For details of the HEX format, refer to appendix 13.1.2, HEX File Format.
stype	Outputs an **S**-type file. For details of the **S**-type format, refer to appendix 13.1.1, S-Type File Format.
binary	Outputs a binary file.

Remarks: Table 5.4 shows relations between output formats and input files or other options.

Table 5.4 Relations Between Output Format And Input File Or Other Options

Output Format	Specified Option	Enabled File Format	Specifiable Option*[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, ps_check, start, rom, entry, output, map, hide, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, section_forbid, absolute_forbid, profile, cachesize, sbr, compress, rename, delete, define, fsymbol, stack, noprelink, memory, msg_unused, data_stuff, show=symbol, reference, xreference, jump_entries_for_pic, aligned_section
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, delete, noprelink, msg_unused, data_stuff, show=symbol, xreference
Object	**extract** specified	Library file	library, output
Hexadecimal Stype Binary		Object file Relocatable file Binary file Library file	input, library, binary, cpu, ps_check, start, rom, entry, output, map, space, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, section_forbid, absolute_forbid, profile, cachesize, sbr, rename, delete, define, fsymbol, stack, noprelink, record, s9*[2], byte_count*[3], memory, msg_unused, data_stuff, show=symbol, reference, xreference, jump_entries_for_pic, aligned_section
		Absolute file	input, output, record, s9*[2], byte_count*[3], show=symbol, reference, xreference
Library	**strip** specified	Library file	library, output, memory*[4], 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*[4], show=symbol, section

Notes: 1. **message**/**nomessage**, **change_message**, **logo**/**nologo**, **form**, **list**, and **subcommand** can always be specified.
2. **s9** can be used only when **form=stype** is specified for the output format.
3. **byte_count** can be used only when **form=hexadecimal** is specified for the output format.
4. **memory** cannot be used when **hide** is specified.

DEBug, SDebug, NODEBug — Debugging Information

Link/Library <Output>[Debug information :]

Format: DEBug

SDebug

NODEBug

Description: Specifies whether debugging information is output.

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

When **sdebug** is specified, debugging information is output to **<output file name>.dbg** file.

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

If **sdebug** and **form=relocate** are specified, **sdebug** is interpreted as **debug**.

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**.

When this option is omitted, the default is **debug**.

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

REcord Record Size Unification

Link/Library <Output>[Data record header :]

Format: REcord = { H16 | H20 | H32 | S1 | S2 | S3 }

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.

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

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

ROm ROM Support Function

Link/Library <Output>[Show entries for :][ROM to RAM mapped sections]

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.

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 Output File

Link/Library <Output> [Show entries for :][Output file path/ Messages] or [Divided output files]

Format: OUtput = <suboption>[,…]

<suboption>: <file name>[=<output range>]

<output range>: {<start address>-<end address> | <section name>[:…]}

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.

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		

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.

MAp Output of External Symbol Allocation Information File

Link/Library <Output>[Generate external symbol-allocation information file]

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.

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

This option is available when the CPU type is SuperH Family or RX Family.

SPace Output to Unused Areas

Link/Library <Output>[Show entries for :][Specify value filled in unused area]
[Output padding data]

Format: SPace [= {<numerical value> | Random}]

Description: Fills the unused areas in the output ranges with random values or a user-specified hexadecimal value.

The following unused areas are filled with the value according to the output range specification in the **output** option:

When section names are specified for the output range:
The specified value is output to unused areas between the specified sections.

When an address range is specified for the output range:
The specified value is output to unused areas within the specified address range.

A 1-, 2-, or 4-byte value can be specified. The hexadecimal value specified to the **space** option determines the output data size. If a 3-byte value is specified, the upper digit is extended with 0 to use it as a 4-byte value. If an odd number of digits are specified, the upper digits are extended with 0 to use it as an even number of digits.

If the size of an unused area is not a multiple of the size of the specified value, the value is output as many times as possible, then a warning message is output.

Remarks: When no suboption is specified by this option, unused areas are not filled with values.

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

When no output range is specified by the **output** option, this option is unavailable.

Message, NOMessage Information Message

Link/Library <Output>[Show entries for :] [Output file path/ Messages]
[Repressed information level messages :]

Format: Message

NOMessage [=<suboption>[,…]]

<suboption>: <error number>[-<error number>]

Description: Specifies whether information level messages are output.

When **message** is specified, information level messages are output.

When **nomessage** is specified, the output of information level messages are 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 (-). 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.

When this option is omitted, the default is **nomessage**.

Examples: `nomessage=4,200-203,1300`

Messages of L0004, L0200 to L0203, and L1300 are disabled to be output.

MSg_unused Notification of Unreferenced Symbol

Link/Library <Output>[Show entries for :] [Output Messages] [Notify unused symbol:]

Format: MSg_unused

Description: Notifies the user of the externally defined symbol which is not referenced during linkage through an output message.

Examples:
```
optlnk -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.

- **goptimize** is not specified at assembly and there are branches to the same section within the same file (only when an H8, H8S, H8SX Family CPU is specified).
- There are references to constant symbols within the same file.
- There are branches to immediate subordinate functions when optimization is specified at compilation.
- The external variable access optimization is valid at compilation (only when an SuperH Family CPU is specified).
- An offset value is directly specified in a **#pragma tbr** in the C source program (only when the SH-2A or SH2A-FPU is specified as the CPU).
- Optimization is specified at linkage and constants or literals are unified.

DAta_stuff Reduce empty areas of boundary alignment

Link/Library <Output>[Show entries for :] [Reduce empty areas of boundary alignment:]

Format: DAta_stuff

Description: At linkage, reduces empty areas of boundary alignment. This option affects constant, initialized and uninitialized data areas.

When this option is specified, empty areas generated as the boundary alignment of sections after compilation are filled at linkage. However, the order of data allocation is not changed.

When this option is not specified, linkage is based on the boundary alignment of sections after compilation.

Specifying this option fills the unnecessary empty areas generated by boundary alignment, reducing the size of the data sections as a whole.

Examples:

```
<tp1.c>          <tp2.c>
-----------------------------
long a;          char d;
char b,c;        long e;
                 char f;
```

Sizes of data sections after compilation (taking the output of the SuperH Family compiler as an example):
tp1.obj: 4 + 1 + 1 = 6 bytes
tp2.obj: 1 + 3 [*] + 4 + 1 = 9 bytes

Sizes of data sections for **tp1.obj** and **tp2.obj** after linkage:
1) When **data_stuff** is not specified
Object files are linked based on the boundary alignment of the sections (conventional process).
6 bytes [tp1] + 2 bytes [*] + 9 bytes [tp2] = 17 bytes

2) When **data_stuff** is specified
Linkage is performed with filling of the unnecessary empty spaces generated between sections by boundary alignment.
(4 + 1 + 1) bytes + 1 byte + 1 byte [*] + 4 bytes + 1 byte = 13 bytes

Notes: 1. * indicates an empty area generated by boundary alignment.

2. The sizes of the data sections after compilation may differ from those in the above example according to the specification of other options, etc. at compilation.

Remarks: Correct operation is not guaranteed if this option is specified when an object file compiled with the **smap** option of the SuperH Family compiler is linked.

The function of this option is not applicable to object files generated by the assembler.

Specification of this option is invalid in any of the following cases:

- **form=library**, **object**, or **relocate** is specified
- An absolute load module is input
- **memory=low** is specified
- **nooptimize** is not specified

Optimization will not be applied in the linkage of a relocatable file that was generated with this option specified.

This option is unavailable when the CPU type is RX Family, M16C Series, or R8C Family.

BYte_count Specification of Data Record Byte Count

Link/Library <Output>[Length of data record :]

Format: BYte_count=<numerical value>

Description: Specifies the maximum byte count for a data record when a file is to be created in the **Intel-Hex** format. Specify a one-byte hexadecimal value (01 to FF) for the byte count. When this option is not specified, the linkage editor assumes FF as the maximum byte count when creating an **Intel-Hex** file.

Examples: `byte_count=10`

Remarks: This option is invalid when the file to be created is not an **Intel-Hex**-type (**form=hex**) file.

CRc CRC

Link/Library <Output> [Show entries for :] [Generate CRC code]

Format: CRc = <suboption>

<suboption>: <address where the result is output>=<target range>
[/<polynomial expression>][:<endian>]

<address where the result is output>: <address>

<target range>: <start address>-<end address>[,...]

<polynomial expression>: { CCITT | 16 }

<endian>: {BIG | LITTLE}

Description: This option is used for cyclic redundancy checking (CRC) of values from the lowest to the highest address of each target range and outputs the calculation result to the specified address.

<endian> can be specified only when the CPU type is RX Family. When <endian> is specified, the calculation result is output to the specified address in the specified endian. When <endian> is not specified, the result is output to the specified address in the endian used in the absolute file.

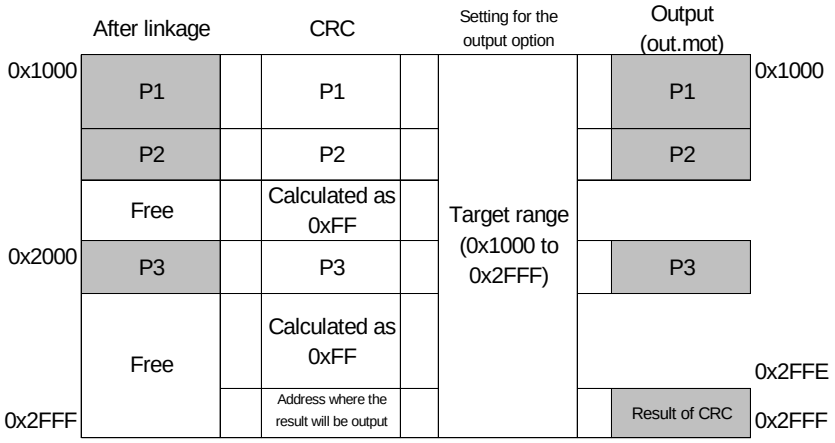
CRC-CCITT or **CRC-16** is selectable as a polynomial expression (default: **CRC-CCITT**).

Polynomial expression:

CRC-CCITT
X^16+X^12+X^5+1
In bit expression: (10001000000100001)

CRC-16
X^16+X^15+X^2+1
In bit expression: (11000000000000101)

Example 1:
```
optlnk *.obj -form=stype -start=P1,P2/1000,P3/2000
       -crc=2FFE=1000-2FFD -output=out.mot=1000-2FFF
```

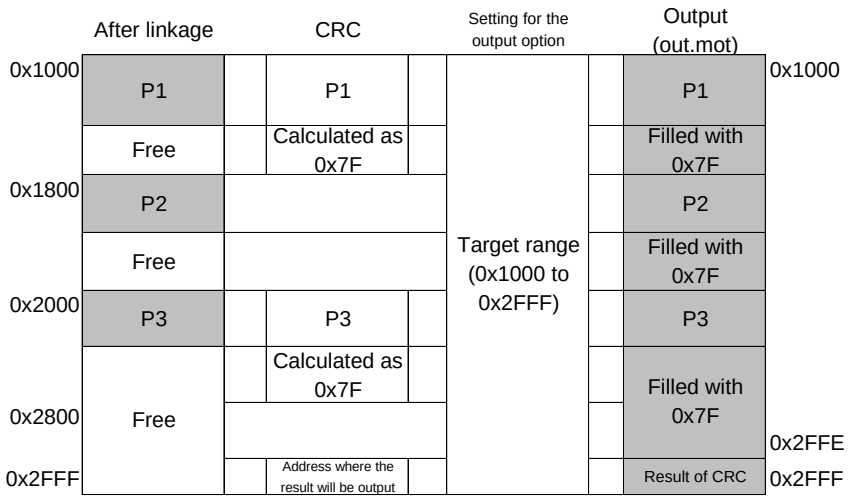


crc option: `-crc=2FFE=1000-2FFD`
In this example, CRC will be calculated for the range from 0x1000 to 0x2FFD and the result will be output to address 0x2FFE.
When the **space** option has not been specified, **space=0xFF** is assumed for calculation of free areas within the target range.

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.
2. The address where the result of CRC will be output must be included in the output range specified with the **output** option.

Example 2:
```
optlnk *.obj -form=stype -start=P1/1000,P2/1800,P3/2000
        -space=7F -crc=2FFE=1000-17FF,2000-27FF
        -output=out.mot=1000-2FFF
```



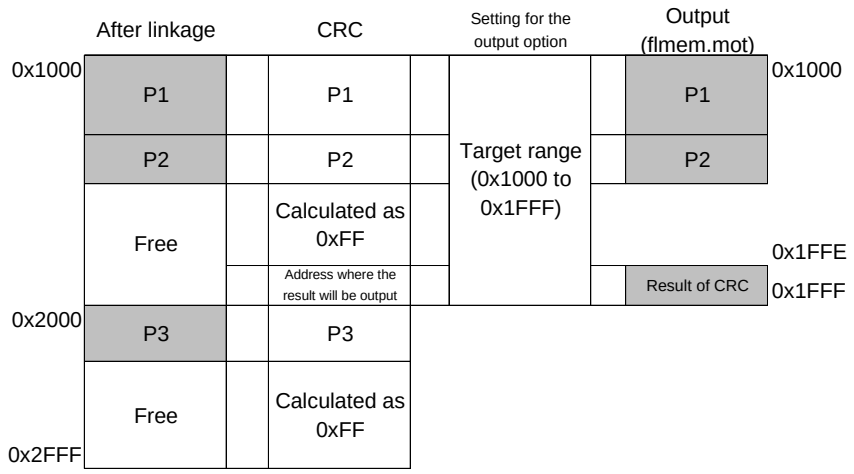
crc option: `-crc=2FFE=1000-2FFD,2000-27FF`
In this example, CRC will be calculated for the two ranges, 0x1000 to 0x17FF and 0x2000 to 0x27FF, and the result will be output to address 0x2FFE.
Two or more non-contiguous address ranges can be selected as the target range for CRC.

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.
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.

Example 3:
```
optlnk *.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.

Remarks: This option is invalid when two or more absolute files have been selected.

This option is valid only when **form={hexadecimal | stype}**.

When the **space** option has not been specified and the target range includes free areas that will not be output, the linkage editor assumes in CRC that 0xFF has been set in the free areas.

An error occurs if the target range includes an overlay area.

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 optlnk.

When the selected polynomial expression is **CRC-CCITT**:

```
typedef  unsigned      char    uint8_t;
typedef  unsigned      short   uint16_t;
typedef  unsigned      long    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 polynomial expression 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 — Filling padding data at section end

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.

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.

This option is valid when the CPU type is SuperH Family or RX Family.

VECTN Address Setting for Specified Vector Number

Link/Library <Output> [Show entries for:] [Address allocation on specific vector]

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>.

Examples:
```
-vectn=30=_f1,31=0000F100 ;Specifies the _f1 address for vector
                          ;number 30 and 0x0f100 for vector
                          ;number 31
```

Remarks: This option is valid when the CPU type is RX Family, M16C Series, or R8C Family.

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 Address Setting for Unused Vector Area

Link/Library <Output> [Show entries for:] [Filling address on empty vector]

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>.

Remarks: This option is valid when the CPU type is RX Family, M16C Series, or R8C Family.

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.

UTL **utl30** information output

Link/Library <Other> [Other option] [utl file output]

Format: UTL

Description: Generates an external file (**utl** file) to be input to the tool (**utl30**) included with the compiler package.

The generated file is assigned a name <output file name>.utl.

Examples:
```
tp.obj
utl
output=test.abs
```

Outputs inspector information from **tp.obj** to **test.utl**.

Remarks: This option is valid only when the compiler for the M16C microcontrollers is used.

This option cannot be used when processing the **abs** files input to the linkage editor.

This option is invalid when **form={object | library}** is specified.

JUMP_ENTRIES_FOR_PIC Jump table output

Link/Library <Output> [Jump table]

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.

This option is used for the PIC function of the RX family compilers.

The file name is <output file>.jmp.

Examples:
```
jump_entries_for_pic=sct2,sct3
output=test.abs
```

A jump table for branching to external definition symbols in the sections **sct2** and **sct3** is output to **test.jmp**.

[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.

This option is invalid when the CPU type is not the RX series.

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.

5.2.3 List Options

Table 5.5 List Category Options

Item	Command Line Format	Dialog Menu	Specification
List file	LISt [= <file name>]	Link/Library <List> [Generate list file]	Specifies the output of list file.
List contents	SHow [= <sub>[,...]] <sub>: {SYmbol | Reference | SEction | Xreference | Total_size| VECTOR| ALL }	Link/Library <List> [Contents :]	 Symbol information Number of references Section information Cross-reference information Total sizes of sections Vector Information All information

LISt List File

Link/Library <List> [Generate list file]

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.

SHow List Contents

Link/Library <List> [Contents]

Format: SHow [=<sub>[,…]]
<sub>:{ SYmbol | Reference | SEction | Xreference | Total_size | VECTOR | ALL}

Description: Specifies output contents of a list.

Table 5.6 lists the suboptions.

For details of list examples, refer to section 7.3, Linkage List, and section 7.4, Library List in the user's manual.

Table 5.6 Suboptions of show Option

Output Format	Suboption Name	Description
form=library or **extract** 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 (when **extract** is specified)
	xreference	Not specifiable
	total_size	Not specifiable
	vector	Not specifiable
	all	Not specifiable (when **extract** is specified) Outputs a symbol name list and a section list in a module (when **form=library**)
Other than **form=library** and **extract** is not 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.
	all	If **form=rel**, the linkage editor outputs the same information as when **show=symbol**, **xreference**, or **total_size** is specified. If **form=rel** and **data_stuff** have been specified, the linkage editor outputs the same information as when **show=symbol** or **total_size** is specified. If **form=abs**, the linkage editor outputs the same information as when **show=symbol**, **reference**, **xreference**, or **total_size** is specified. If **form=hex**, **stype**, or **bin**, the linkage editor outputs the same information as when **show=symbol**, **reference**, **xreference**, or **total_size** 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	Reference	Section	Xreference	Vector	Total_size
form=abs	show	Valid	Valid	Invalid	Invalid	Invalid	Invalid
	show=all	Valid	Valid	Invalid	Valid	Valid	Valid
form=lib	show	Valid	Invalid	Valid	Invalid	Invalid	Invalid
	show=all	Valid	Invalid	Valid	Invalid	Invalid	Invalid
form=rel	show	Valid	Invalid	Invalid	Invalid	Invalid	Invalid
	show=all	Valid	Invalid	Invalid	Valid*	Invalid	Valid
form=obj	show	Valid	Valid	Invalid	Invalid	Invalid	Invalid
	show=all	Valid	Invalid	Invalid	Invalid	Invalid	Invalid
form=hex/bin/sty	show	Valid	Valid	Invalid	Invalid	Invalid	Invalid
	show=all	Valid	Valid	Invalid	Valid	Valid*	Valid*

Note: The option is invalid if an absolute-format file is input.

Note the following limitations on output of the cross-reference information.

- When the relocatable format is specified for the output file and the **data_stuff** option is specified, no cross-reference information is output.
- When an absolute-format file is input, the referrer address information is not output.
- When **-goptimize** is not specified at assembly, information about branches to the same section within the same file is not output (only when an H8, H8S, H8SX Family CPU is specified).
- 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 (only when an SuperH Family or RX Family CPU is specified).
- When an offset value is directly specified in a **#pragma tbr** in the C source program, information about that function is not output (only when the SH-2A or SH2A-FPU is specified as the CPU).

- When optimization is specified at linkage and constants or literals are unified, information about references to these constants or literals is not output.
- Both **show=total_size** and **total_size** output the same information.
- **show=vector** can be used when the CPU type is RX Family, M16C Series, or R8C Family.
- When **show=reference** is valid, the number of references of the variable specified by **#pragma address** is output as 0 (only when a SuperH Family or RX Family CPU is specified).

5.2.4 Optimize Options

Table 5.7 Optimize Category Options

Item	Command Line Format	Dialog Menu	Specification
Optimization	OPtimize = <sub>[…] <sub>: {STring_unify | SYmbol_delete | Variable_access | Register | SAMe_code | SHort_format | Function_call | Branch | Speed | SAFe } NOOPtimize}	Link/Library <Optimize> [Show entries for :] [Optimize items] [Optimize :]	Executes optimization. Unifies constants/string literals. Deletes unreferenced symbols. Uses short absolute addressing mode. Provides optimization with register save/restore. Unifies same codes. Shortens the addressing mode. Uses indirect addressing mode. Provides optimization for branches. Provides optimization for speed. Provides safe optimization. No optimization.
Same code size	SAMESize = <size> (default: sames=1e)	Link/Library <Optimize> [Eliminated size :]	Specifies the minimum size to unify same codes.
Profile information	PROfile = <file name>	Link/Library <Optimize> [Include profile :]	Specifies a profile information file. (Dynamic optimization is provided.)
Cache size	CAchesize=<sub> <sub>: Size=<size> | Align=<line size> (default: ca=s=8,a=20)	Link/Library <Optimize> [Cache size :]	 Specifies a cache size. Specifies a cache line size. (for SuperH Family)

Item	Command Line Format	Dialog Menu	Specification
Optimization partially disabled	SYmbol_forbid= <symbol name>[,…]	Link/Library <Optimize> [Show entries for :] [Forbid item]	Specifies a symbol where unreferenced symbol deletion is disabled.
	SAMECode_forbid= <function name>[,…]		Specifies a symbol where same code unification is disabled.
	Variable_forbid= <symbol name>[,…]		Specifies a symbol where short absolute addressing mode is disabled.
	FUnction_forbid= <function name>[,…]		Specifies a symbol where indirect addressing mode is disabled.
	SEction_forbid = <sub>[,...] <sub>: [<file name>| <module name>] (<section name>[,...])		Specifies a section where optimization is disabled.
	Absolute_forbid= <address>[+<size>][,…]		Specifies an address range where optimization is disabled.

OPtimize, NOOPtimize Optimization

Link/Library <Optimize> [Show entries for :][Optimize items][Optimize :]

Format: OPtimize [= <suboption>[,…]]

NOOPtimize

<suboption>: { STring_unify | SYmbol_delete | Variable_access | Register | SAMe_code | SHort_format | Function_call | Branch | SPeed | SAFe }

Description: Specifies whether the inter-module optimization is executed.

When **optimize** is specified, optimization is performed for the file specified with the **goptimize** option at compilation or assembly.

When **nooptimize** is specified, no optimization is executed for a module.

When this option is omitted, the default is **optimize**.

Table 5.8 shows the suboptions

Table 5.8 Suboptions of Optimize Option

		Program to be Optimized*[1]							
Suboption	**Description**	**SHC**	**SHA**	**H8C**	**H8A**	**RXC**	**RXA**	**NCC**	**NCA**
No parameter	Provides all optimizations	O	×	O	O	O	×	O	×
string_unify	Unifies same-value constants having the **const** attribute. Constants having the **const** attribute are: • Variables defined as **const** in C/C++ program • Initial value of character string data • Literal constant	O	×	O	×	×	×	×	×

		Program to be Optimized*1							
Suboption	**Description**	**SHC**	**SHA**	**H8C**	**H8A**	**RXC**	**RXA**	**NCC**	**NCA**
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	×	O	×	O	×	O	×
variable_access	Allocates frequently accessed variables to the area accessible in the 8/16 bit absolute addressing mode. The **cpu** option should be specified at compilation and assembly.	×	×	O	O	×	×	×	×
register	Investigates function calls, relocates registers and deletes redundant register save or restore codes. Always be sure to specify **#pragma entry** at compilation or the **entry** option in the optimizing linkage editor.	O	×	O	×	×	×	×	×
same_code	Creates a subroutine for the same instruction sequence.	O	×	O	×	O	×	×	×
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	O	×	×	×
function_call	Allocates addresses of frequently accessed functions to the range 0 to 0xFF if there is a space. When the CPU is H8SX Family, the following ranges are also used: H8SXN: 0x100 to 0x1FF H8SXM,H8SXA,H8SXX: 0x200 to 0x3FF The **cpu** option should be specified at compilation and assembly.	×	×	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	O	O	×	O	×

Suboption	Description	Program to be Optimized*1							
		SHC	SHA	H8C	H8A	RXC	RXA	NCC	NCA
speed	Executes optimizations other than those reducing object speed. This suboption is the same as the following specifications: optimize=string_unify, symbol_delete, variable_access, register, short_format, or branch	O	×	O	O	O*2	×	O*2	×
safe	Executes optimizations other than those limited by variable or function attributes. This suboption is the same as the following specifications: optimize=string_unify, register, short_format, or branch	O	×	O	O	O*4	×	O*3	×

Notes: 1. SHC: C/C++ program for SuperH Family
SHA: Assembly program for SuperH Family
H8C: C/C++ program for H8, H8S, H8SX Family
H8A: Assembly program for H8, H8S, H8SX Family
RXC: C/C++ program for RX Family,
RXA: Assembly program for RX Family
NCC: C/C++ program for M16C Series or R8C Family
NCA: Assembly program for M16C Series or R8C Family

2. **symbol_delete**, **branch**, and **short_format** are valid in optimization for which **speed** was specified.
3. **branch** is valid in optimization for which **safe** was specified.
4. s**hort_format** and **branch** are valid in optimization for which **safe** was specified.

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

When optimization of access to external variables is specified at compilation, optimization with unification of constants/string literals (**optimize=string_unify**) is invalid.

optimize=short_format is available only when the CPU is H8SX Family or RX Family.

When the CPU is SH-2A or SH2A-FPU, the code size may increase due to the **optimize=register** function.

When a start function with **#pragma entry** or **entry** is not specified, **optimize=symbol_delete** is invalid.

SAMesize Common Code Size

Link/Library <Optimize> [Eliminated size :]

Format: SAMESize = <size>

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.

When this option is omitted, the default is **samesize=1E**.

Remarks: When **optimize=same_code** is not specified, this option is unavailable.

PROfile Profile Information

Link/Library <Optimize> [Include profile :]

Format: PROfile = <file name>

Description: Specifies a profile information file.

Specifiable profile information files are those output from the High-performance Embedded Workshop Ver. 2.0 or later.

When a profile information file is specified, inter-module optimization according to dynamic information can be performed.

Table 5.9 shows optimizations influenced by a profile information input.

Table 5.9 Relations Between Profile Information and Optimization

		Program to be Optimized*[1]			
Suboption	**Description**	**SHC**	**SHA**	**H8C**	**H8A**
variable_access	Allocates variables from those that are dynamically accessed more frequently.	×	×	O	O
function_call	Lowers the optimizing priority of functions that are dynamically accessed frequently.	×	×	O	O
branch	Allocates a function that is dynamically accessed frequently near the calling function. For the SH program, the optimization with allocation is performed depending on the cache size specified using the **cachesize** option.	O	Δ*[2]	O	Δ

Notes: 1. SHC: C/C++ program for SuperH Family
SHA: Assembly program for SuperH Family
H8C: C/C++ program for H8, H8S, H8SX Family
H8A: Assembly program for H8, H8S, H8SX Family
2. Movement is provided not in the function unit, but in the input file unit.

Remarks: When the **optimize** option is not specified, this option is unavailable.

CAchesize Cache Size

Link/Library <Optimize> [Cache size :]

Format: CAchesize = <suboption>

<suboption>: Size = <size> | Align = <line size>

Description: Specifies a cache size and cache line size.

When **profile** is specified, this option is used at the branch instruction optimization (**optimize=branch**).

Specify the size in Kbytes and specify the line size in bytes in the hexadecimal notation.

When this option is omitted, the default is **cachesize=size=8, align=20**.

Remarks: If **profile** is not specified, this option is unavailable.

SYmbol_forbid, SAMECode_forbid, Variable_forbid, FUnction_forbid, SEction_forbid, Absolute_forbid Optimization Partially Disabled

Link/Library <Optimize> [Show entries for :] [Forbid item]

Format: SYmbol_forbid = <symbol name> [,…]

SAMECode_forbid = <function name> [,…]

Variable_forbid = <symbol name> [,…]

FUnction_forbid = <function name> [,…]

SEction_forbid = <sub>[,…]

<sub>: [<file name>|<module name>](<section name>[,…])

Absolute_forbid = <address> [+<size>] [,…]

Description: Disables optimization for the specified symbol, section, or address range. Specify an address or the size in the hexadecimal notation. 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>( )".

Table 5.10 shows the suboptions.

Table 5.10 Suboptions of Optimization Partially Disabling Option

Suboption	Parameter	Description
symbol_forbid	Function name | variable name	Disables optimization regarding unreferenced symbol deletion
samecode_forbid	Function name	Disables optimization regarding same-code unification
variable_forbid	Variable name	Disables optimization regarding short absolute addressing mode
function_forbid	Function name	Disables optimization regarding indirect addressing mode
section_forbid	Section name File name Module name	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.
absolute_forbid	Address [+ size]	Disables optimization regarding address + size specification

Examples:

```
symbol_forbid="f(int)"          ; Does not delete the C++ function f(int)
                                ; even if it is not referenced.

section_forbid=(P1)             ; Disables any optimization for section
                                ; P1.

section_forbid=a.obj(P1,P2)     ; Disables any optimization for sections
                                ; P1 and P2 in a.obj.
```

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**.

5.2.5 Section Options

Table 5.11 Section Category Options

Item	Command Line Format	Dialog Menu	Specification
Section address	STARt = <sub>[,…] <sub>: [(]<section name> [{ : | , }<section name>[,…]] [)][,...] [/<address>]	Link/Library <Section> [Show entries for :] [Section]	Specifies a section start address
Symbol address file	FSymbol = <section name>[,…]	Link/Library <Section> [Show entries for :] [Symbol file]	Outputs externally defined symbol addresses to a definition file.
Section alignment specification	ALIGNED_SECTION = <section name>[,...]	Link/Library <Section> [Show entries for :] [Section alignment]	Changes the section alignment value to 16 bytes.

STARt Section Address

Link/Library <Section> [Show entries for :] [Section]

Format: STARt = <sub> [,…]

<sub>: [(] <section name> [{ : | , } <section name> [,…]] [)] [,…]
[/ <address>]

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.

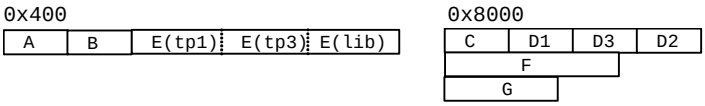
If no address is specified, the section is allocated at 0.

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).

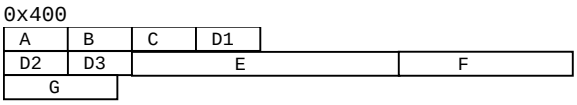
```
tp1.obj(A,D1,E) -> tp2.obj(B,D3,F)) -> tp3.obj(C,D2,E,G)
 -> lib.lib(E)
```

(1) `-start=A,B,E/400,C,D*:F:G/8000`



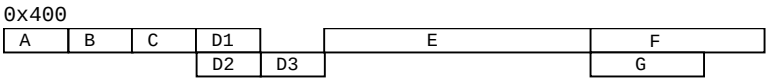
- Sections **C**, **F**, and **G** separated by colons are allocated to the same address.
- Sections specified with wildcards “*” (in this example, the sections whose names start with **D**) are allocated in the input order.
- Objects in the sections having the same name (**E** in this example) are allocated 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.

(2) `-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.

(3) `-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 Symbol Address File

Link/Library <Section> [Show entries for :][Symbol file]

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:
```
fSymbol = sct2, sct3
output=test.abs
```

Outputs externally defined symbols in sections **sct2** and **sct3** to **test.fsy**.

[Output example of **test.fsy**]
```
;OPTIMIZING LINKAGE EDITOR GENERATED FILE 1999.11.26
;fsymbol = sct2, sct3

;SECTION NAME = sct2
 .export _f
_f: .equ h’00000000
 .export _g
_g: .equ h’00000016
;SECTION NAME = sct3
 .export _main
_main: .equ h’00000020
 .end
```

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

This option is available when the CPU type is H8, H8S, H8SX Family, SuperH Family, or RX Family.

ALIGNED_SECTION Changing Section Alignment to 16 bytes

Link/Library <Section> [Show entries for :][Section alignment]

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.

5.2.6 Verify Options

Table 5.12 Verify Category Options

Item	Command Line Format	Dialog Menu	Specification
Address check	CPu = { <cpu information file name> | <memory type> = <address range>[,…] | STRIDE} <memory type>: { ROm | RAm | XROm | XRAm | YROm | YRAm } <address range>: <start address> -<end address>	Link/Library <Verify> [CPU information check :]	Specifies a specifiable allocation range for section addresses. The specified section will be divided.
Physical space overlap check	PS_check=<sub>[:<sub>...] <sub>: <LS>,<LS>[,...] <LS>: <start address> -<end address>	Link/Library <Verify> [Physical space overlap check :]	Specifies address ranges that may overlap each other in the physical space.
Not divide the specified section	CONTIGUOUS_SECTION = <section name>[,…]	Link/Library <Verify> [Not divide the specified section :]	The specified section will not be divided.

CPu Address Check

Verify [CPU information check:]

Format: CPu={<cpu information file name>
| <memory type> = <address range> [,…]
| STRIDE}

<memory type>: { ROm | RAm | XROm | XRAm | YROm | YRAm | 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.

[Example]
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.

[Example]
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.

xrom and **xram** specify the X memory areas and **yrom** and **yram** specify the Y memory areas in the DSP.

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.

[Example]
```
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.

Memory types **xrom**, **xram**, **yrom**, and **yram** are available only when the CPU is SHDSP, SH2DSP, SH3DSP or SH4ALDSP.

When **cpu=stride** and **optimize=register** are valid, error L2320 may be output. In such cases, disable **optimize=register**.

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.

PS_check Physical Space Overlap Check

Verify [Physical space overlap check :]

Format: PS_check=<sub>[:<sub>...]

<sub>: <LS>,<LS>[,...]

<LS>: <start address>-<end address>

Description: Specifies objects that may overlap each other when they are allocated to the memory.

Use this option to detect SH3 or SH4 objects that will overlap each other when they are allocated to the actual memory even if their virtual addresses do not overlap.

If an overlap is detected after this option setting, an error will be output and the linkage operation will be terminated.

Specify address ranges (<LS> in the command line format) that may overlap each other in the memory.

To check multiple physical memory spaces, specify them by separation with a colon (:).

Examples: In the SH4, the 4-Gbyte address space is mapped to the 512-Mbyte (29-bit address) external memory area when the MMU is disabled (the upper three bits of address for the 4-Gbyte space are ignored).

For example, when the **U0** area (00000000 to 0x7fffffff) that can be used in user mode is mapped to the external memory (512 Mbytes), overlapped objects can be detected through the following setting.

```
-PS_check=00000000-1fffffff,20000000-3fffffff,
40000000-5fffffff,60000000-7fffffff
```

This setting means that addresses 00000000, 20000000, 40000000, and 60000000 are allocated to the same location in the actual memory.

Remarks: This option is only valid for the SuperH Family CPUs.

This option is invalid if **object**, **relocate**, or **library** is specified for the **output** format (**form** option).

This option is invalid when an absolute file is input.

For the address space specifications of the CPU, refer to the hardware manual of the target CPU.

CONTIGUOUS_SECTION — Not divide the specific section

Link/Library <Verify> [Not divide the specified section :]

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.

5.2.7 Other Options

Table 5.13 Other Category Options

Item	Command Line Format	Dialog Menu	Specification
End code	S9	Link/Library <Other> [Miscellaneous options :] [Always output S9 record at the end]	Always outputs the S9 record.
Stack information file	STACk	Link/Library <Other> [Miscellaneous options :] [Stack information output]	Outputs a stack use information file.
Debugging information compression	Compress NOCOmpress	Link/Library <Other> [Miscellaneous options :] [Compress debug information]	Compresses debugging information Does not compress debugging information
Memory occupancy reduction	MEMory = [High | Low]	Link/Library <Other> [Miscellaneous options :] [Low memory use during linkage]	Specifies the memory occupancy when an input file is loaded
Symbol name modification	REName = <sub>[,…] <sub>: {<file name> (<name>=<name>[,…]) | <module name> (<name><name>[,…]) }	Link/Library <Other> [User defined options :]	Modifies a symbol name or section name.
Symbol name deletion	DELete = <sub>[,…] <sub>: {<module name> | [<file name>] (<name>[,…]) }	Link/Library <Other> [User defined options :]	Deletes a symbol name or module name.
Module replacement	REPlace = <sub>[,…] <sub>: <file> [(<module>[,…])]	Link/Library <Other> [User defined options :]	Replaces modules of the same name in a library file.
Module extraction	EXTract = <module>[,…]	Link/Library <Other> [User defined options :]	Extracts the specified module in a library file.
Debugging information deletion	STRip	Link/Library <Other> [User defined options:]	Deletes debugging information in an absolute file or a library file.

Item	Command Line Format	Dialog Menu	Specification
Message level	CHange_message=<sub>[,…] <sub>: {Information | Warning | Error } [=<error number> [-<error number>] [,…]]	Link/Library <Other> [User defined options:]	Modifies message levels.
Local symbol name hide	Hide	Link/Library <Other> [User defined options:]	Deletes local symbol name information
Showing total sizes of sections	Total_size	Link/Library <Other> [Miscellaneous options :] [Displays total section size]	This newly added option sends total sizes of sections after linkage to standard output.
Information file for the emulator	RTs_file	Link/Library <Other> [Miscellaneous options :] [Rts information output]	Outputs an information file for the emulator (for SuperH Family).

S9 End Code

Link/Library <Other>[Miscellaneous options :][Always output S9 record at the end]

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 Stack Information File

Link/Library <Other>[Miscellaneous options :][Stack information output]

Format: STACk

Description: Outputs a stack consumption 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, NOCOmpress Debugging Information Compression

Link/Library <Other>[Miscellaneous options :][Compress debug information]

Format: COmpress

NOCOmpress

Description: Specifies whether debugging information is compressed.

When **compress** is specified, the debugging information is compressed.

When **nocompress** is specified, the debugging information is not compressed.

By compressing the debugging information, the debugger loading speed is improved. If the **nocompress** option is specified, the link time is reduced.

If this option is omitted, the default is **nocompress**.

Remarks: When **form={object | relocate | library | hexadecimal | stype | binary}** or **strip** is specified, the compress option is unavailable.

MEMory Memory Occupancy Reduction

Link/Library <Other>[Miscellaneous options :][Low memory use during linkage]

Format: MEMory = [High | Low]

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**, or **replace**

When **form=object** or **relocate** is specified:

extract

When the microcontroller is of a type that is not a member of the NC family and **optimize** is specified.

Some combinations of this option and the input or output file format are unavailable. For details, refer to table 5.4 in section 5.2.2, Output Options.

REName Symbol Name Modification

Link/Library <Other>[User defined options :]

Format: REName = <suboption> [,…]

<suboption>: {[<file>] (<name> = <name> [,…])
| [<module>] (<name> = <name> [,…]) }

Description: Modifies a 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.

DELete Symbol Name Deletion

Link/Library <Other>[User defined 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.

REPlace Module Replacement

Link/Library <Other>[User defined 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.

EXTract Module Extraction

Link/Library <Other>[User defined 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 Debugging Information Deletion

Link/Library <Other>[User defined 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 Message Level

Link/Library <Other>[User defined 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.

Examples: `change_message=warning=2310`
Modifies L2310 to the warning level and specifies execution continuation at L2310 output.

`change_message=error`
Modifies all information and warning messages to error level messages. When a message is output, the execution is aborted.

Hide Local Symbol Name Hide

Link/Library <Other>[User defined 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 Showing total sizes of sections

Link/Library <Other> [Miscellaneous options :] [Displays total section size]

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-support function (**rom** option) has been specified for a section, the section will be used by both the source (ROM) and destination (RAM) of the transfer. The sizes of sections of this type will be added to the total sizes of sections in both ROM and RAM.

RTs_file Information File for the Emulator

Link/Library <Other> [Miscellaneous options :] [Rts information output]

Format: RTs_file

Description: This option creates a return address information file (**.rts** file) for the emulator. For usage of this option, refer to the user's manual for the emulator in use. This option is not available in some types of emulators.

The name of the return address information file is **<load module name>.rts**. If the file to be output is **test.abs** as specified with the **output** option, for example, its file will be created as **test.rts**. The return address information file is created under the same directory where the load module has been created.

Remarks: This option is invalid when **form={object | relocate | library}** has been specified.

This option is invalid when an absolute file is selected as an input file.

For usage of this option, refer to the user's manual for the emulator in use. This option is not available in some types of emulators.

This option can be used when the CPU type is SuperH Family.

5.2.8 Subcommand File Options

Table 5.14 Subcommand Tab Option

Item	Command Line Format	Dialog Menu	Specification
Subcommand file	SUbcommand = <file name>	Link/Library <Subcommand file> [Use external subcommand file]	Specifies options with a subcommand file

SUbcommand — Subcommand File

Link/Library <Subcommand file> [Use external subcommand file]

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 (&).

The **subcommand** option cannot be specified in the subcommand file.

Examples: Command line specification:

```
optlnk 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**.

5.2.9 CPU Option

Table 5.15 CPU Tab Option

Item	Command Line Format	Dialog Menu	Specification
SBR address specification	SBr = { <SBR address> | User}	CPU [Specify SBR address :]	Specifies the start address of the 8-bit absolute area (for H8SX Family).

SBr SBR Address Specification

CPU [Specify SBR address :]

Format: SBr = { <address> | User }

Description: Specifies the **SBR** address.

When the **SBR** address is specified in this option, optimization using the **abs8** area is available. When **user** is specified in this option, optimization for the **abs8** area is disabled.

Remarks: This option is available only when the CPU is H8SX Family.

If more than one **SBR** address is specified within the source or by tool options, the optimizing linkage editor assumes that **user** is specified regardless of this option setting.

5.2.10 Options Other Than Above

Table 5.16 Options Other Than Above

Item	Command Line Format	Dialog Menu	Specification
Copyright	LOgo NOLOgo	— (NOLOgo is always valid)	Output Not output
Continuation	END	—	Executes option strings already input, inputs continuing option strings and continues processing.
Termination	EXIt	—	Specifies the termination of option input.

LOgo, NOLOgo Copyright

None (nologo is always available.)

Format: LOgo

NOLOgo

Description: Specifies whether the copyright is output.

When the **logo** option is specified, the copyright is displayed.

When the **nologo** option is specified, the copyright display is disabled.

When this option is omitted, the default is **logo**.

END Execution Continued

None

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 Termination Processing

None

Format: EXIt

Description: Specifies the end of the option specifications.

This option cannot be specified on the command line.

Examples: Command line specification:

```
optlnk -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.

Section 6 Environment Variables

6.1 Environment Variables

Environment variables are listed in table 6.1.

Table 6.1 Environment Variables

No.	Environment Variable	Description	Default When Specification is Omitted
1	path	Specifies a storage directory for the execution file.	Specification cannot be omitted.
2	BIN_RX	Specifies the directory in which **ccrx** is stored.	<ccrx storage directory> Specification cannot be omitted when the **lbgrx** command is used.
3	CPU_RX	Specifies the CPU type. <CPU type> RX600 RX200	No value is set when 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 specification is omitted.
6	TMP_RX	Specifies a directory in which a temporary file is generated.	**%TEMP%** when the **ccrx** command is used.
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 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 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 specification is omitted.

When the **cpu** option does not select the CPU type, the environment variable **CPU_RX** must be set. When a CPU other than **RX600** or **RX200** is specified as the CPU type, an error will occur. For the relationship between **CPU_RX** and the **cpu** option, refer to the description of the **cpu** option in section 2.5, Microcontroller Options.

When more than one directory is specified by **INC_RX**, **INC_RXA**, **HLNK_LIBRARY1**, **HLNK_LIBRARY2**, **HLNK_LIBRARY3**, and **HLNK_DIR**, the directories should be divided using semicolons (;).

For **BIN_RX**, **INC_RX**, and **TMP_RX** when the **ccrx** command is executed, if an environment variable has already been specified, the specified value is used. If no specification has been made, the default at specification omission will be used.

These environment variables can be set easily by executing the batch file **setccrx.bat** which is generated at installation. **setccrx.bat** is stored in "**<High-performance Embedded Workshop storage directory>\Tools\Renesas\RX\1_0_0**" or "**<ccrx storage directory>\..**".

6.2 Predefined Macros

The following predefined macros are defined according to the option specification and version.

Table 6.2 Predefined Macros of Compiler

No.	Option	Predefined Macro	
1	cpu=rx600	#define __RX600	1
	cpu=rx200	#define __RX200	1
2	endian=big	#define __BIG	1
	endian=little	#define __LIT	1
3	dbl_size=4	#define __DBL4	1
	dbl_size=8	#define __DBL8	1
4	int_to_short	#define __ INT_SHORT	1
5	signed_char	#define __SCHAR	1
	unsigned_char	#define __UCHAR	1
6	signed_bitfield	#define __SBIT	1
	unsigned_bitfield	#define __UBIT	1
7	round=zero	#define __ROZ	1
	round=nearest	#define __RON	1
8	denormalize=off	#define __DOFF	1
	denormalize=on	#define __DON	1
9	bit_order=left	#define __BITLEFT	1
	bit_order=right	#define __BITRIGHT	1
10	auto_enum	#define __AUTO_ENUM	1
11	library=function	#define __FUNCTION_LIB	1
	library=intrinsic	#define __INTRINSIC_LIB	1
12	fpu	#define __FPU	1
13	—	#define __RENESAS__*[1]	1
14	—	#define __RENESAS_VERSION__*[1]	0xAABBCC00*[2]
15	—	#define __RX*[1]	1
16	pic	#define __PIC	1
17	pid	#define __PID	1

Notes: 1. Always defined regardless of the option.
2. When the version is V.AA.BB.CC, the value of __**RENESAS_VERSION**__ is 0xAABBCC00.
Example: For V.1.01.00, specify **#define __RENESAS_VERSION__0x01010000**.

Table 6.3 Predefined Macros of Assembler

No.	Option	Predefined Macro	
1	cpu=rx600	__RX600	.DEFINE 1
	cpu=rx200	__RX200	.DEFINE 1
2	endian=big	__BIG	.DEFINE 1
	endian=little	__LITTLE	.DEFINE 1
3	—	__RENESAS_VERSION__*[1]	.DEFINE AABBCC00H*[2]
4	—	__RX*[1]	.DEFINE 1

Notes: 1. Always defined regardless of the option.
2. When the version is V.AA.BB.CC, the value of __**RENESAS_VERSION**__ is 0xAABBCC00.
Example: For V.1.01.00, specify __**RENESAS_VERSION**__ **.DEFINE 01010000H**.

Section 7 File Specifications

7.1 Naming Files

A standard file extension is automatically added to the name of a compiled file when omitted. The standard file extensions used by the integrated development environment are shown in table 7.1.

Table 7.1 Standard File Extensions Used by the Integrated Development Environment

No.	File Extension	Description
1	c	Source program file written in C
2	cpp, cc, cp	Source program file written in C++
3	h	Include file
4	p	C source program preprocessor expansion file
5	pp	C++ source program preprocessor expansion file
6	src	Assembly source program file
7	lst	Assembly program list file
8	obj	Relocatable object program file
9	abs	Absolute load module file
10	map	Linkage map list file
11	lib	Library file
12	lbp	Library list file
13	mot	S-type format file
14	hex	HEX format file
15	bin	Binary file
16	sni	Stack information file
17	pro	Profile information file
18	dbg	Debugging information file
19	rti	Object file including definition that is specified by a file with extension **td**
20	cal	Information file to be called
21	bls	Information file for external symbol allocation
22	jmp	Jump table file (assembly language)
23	fsy	Symbol address file (assembly language)

Note: Conditions for file names
A file name accepted by the compiler should satisfy the following conditions.
- Satisfies the naming conventions of the OS.
- Does not start with a hyphen (-).

File names beginning with **rti_** are reserved for the system; do not use those file names.

Table 7.2 lists the extensions for files that are temporarily output under the **tpldir** folder.

Table 7.2 tpldir Folder Output File

No.	File Extension	Description
1	td	Tentative-defined variable information file
2	ti	Template information file
3	pi	Parameter information file
4	ii	Instance information file

7.2 Source List

7.2.1 Structure of Source List

The source list file contains the compilation and assembly results. Table 7.3 shows the structure and contents of the source list.

Table 7.3 Structure and Contents of Source List

No.	Output Information	Contents	Suboption *	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.

7.2.2 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 section 7.2.3, Object Information.

7.2.3 Object Information

Figure 7.1 shows an example of object information output.

```
* RX FAMILY ASSEMBLER V.1.00.00 *  SOURCE LIST  Sat May 16 11:56:15 2009

LOC.     OBJ.              0XMDA SOURCE STATEMENT
(1)      (2)                (3)       (4)

                                    ;C LABEL    INSTRUCTION OPERAND    COMMENT
                                       (5)          (6)                 (7)
                                    ;LineNo. C-SOURCE STATEMENT
                                       (8)          (9)

                                             .SECTION    P,CODE
                                    ;       1 #include     "include.h"
                                    ;       1 extern int   x;
                                    ;       2 extern int   y = 1;
                                    ;       2 int func01(int);
                                    ;       3 int func03(int);
                                    ;       4
                                    ;       5 int func02(int z)
                                             .glb        _func02
00000000                            _func02:                           ; function: func02
                                             .STACK      _func02=8
00000000 7EA6                                PUSH.L      R6
00000002                            L10:
                                             .LINE       "D:\RXC\work\list\now\sample.c",7
                                    ;       6 {
                                    ;       7       x = func01(z);
00000002 EF16                                MOV.L       R1,R6
00000004 05rrrrrr             A              BSR         _func01
00000008 FB42rrrrrrrr                        MOV.L       #_x,R4
0000000E E341                                MOV.L       R1,[R4]
                                             .LINE       "D:\RXC\work\list\now\sample.c",8
                                    ;       8       if (z == 2) {
00000010 6126                                CMP         #02H,R6
00000012 18                   S              BNE         L12
00000013                            L11:
                                             .LINE       "D:\RXC\work\list\now\sample.c",9
                                    ;       9               x++;
00000013 711501                              ADD         #01H,R1,R5
00000016 E345                                MOV.L       R5,[R4]
00000018 2E11                 B              BRA         L13
0000001A                            L12:
                                             .LINE       "D:\RXC\work\list\now\sample.c",11
                                    ;      10       } else {
                                    ;      11               x = func03(x + 2);
0000001A 6221                                ADD         #02H,R1
0000001C 391200               W              BSR         _func03
0000001F FB42rrrrrrrr                        MOV.L       #_x,R4
00000025 E341                                MOV.L       R1,[R4]
00000027 EF15                                MOV.L       R1,R5
00000029                            L13:
                                             .LINE       "D:\RXC\work\list\now\sample.c",13
                                    ;      12       }
                                    ;      13       return x;
00000029 EF51                                MOV.L       R5,R1
0000002B 3F6601                              RTSD        #04H,R6-R6
                                             .LINE       "D:\RXC\work\list\now\sample.c",16
                                    ;      14 }
                                    ;      15
                                    ;      16 int func03(int p)
                                             .glb        _func03
0000002E                            _func03:                           ; function: func03
                                             .STACK      _func03=4
0000002E                            L14:
                                             .LINE       "D:\RXC\work\list\now\sample.c",18
                                    ;      17 {
                                    ;      18       return p+1;
0000002E 6211                                ADD         #01H,R1
00000030 02                                  RTS
                                    ;      19 }
                                             .glb        _x
                                             .glb        _func01
                                             .SECTION    D,ROMDATA,ALIGN=4
                                             .glb        _y
00000000                            _y:                                ; static: y
00000000 01000000                            .lword       00000001H
                                             .END
```

Figure 7.1 Example of Object Information Output

(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.

(3) Line information (0XMDA)

Results of source code processing by the assembler. The following shows the meaning of each symbol.

Table 7.4 Line Information on Assembly-Language Source Code

0	X	M	D	A	Description
0-30					Shows the nesting level of include files.
	X				Shows the line where the condition is false in conditional assembly when **-show=conditions** is specified.
		M			Shows the line expanded from a macro instruction when **-show=expansions** is specified.
		D			Shows the line that defines a macro instruction when **-show=definitions** is specified.
			S		Shows that branch distance specifier **S** is selected.
			B		Shows that branch distance specifier **B** is selected.
			W		Shows that branch distance specifier **W** is selected.
			A		Shows that branch distance specifier **A** is selected.
				*	Shows that a substitute instruction is selected for a conditional branch instruction.

(4) Source information (SOURCE STATEMENT)

Contents of the assembly-language source file.

(5) Label information (C LABEL)

(6) Assembly-language instructions (INSTRUCTION OPERAND)

Assembly-language instructions output by the compiler.

(7) Comment on assembly-language source program (COMMENT)

(8) C/C++ source line number (LineNo.)

(9) C/C++ source statement (C-SOURCE STATEMENT)

C/C++ source statement output when the **show=source** option is specified.

7.2.4 Statistics Information

Figure 7.2 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
```

Figure 7.2 Example of Statistics Information Output

(1) Numbers of error messages and warning messages, and total number of source lines
(2) Section information (section attribute, size, and section name)

7.2.5 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. Figure 7.3 shows an example of command specification information output.

```
;*** CPU TYPE ***  (1)

;-CPU=RX600

;*** COMMAND PARAMETER ***  (2)

;-output=src=C:\tmp\elp1894\sample.src
;-nologo
;-show=source
;sample.c
```

Figure 7.3 Example of Command Specification Information Output

(1) Selected microcomputer

(2) File names and options specified for the compiler

7.2.6 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. Figure 7.4 shows an example of command specification information output.

```
Cpu Type   (1)

-CPU=RX600

Command Parameter   (2)

-output=sample.obj
-nologo
-listfile=sample.lst
```

Figure 7.4 Example of Command Specification Information Output

(1) Microcomputer selected for the assembler

(2) File names and options specified for the assembler

7.3 Linkage List

This section covers the contents and format of the linkage list output by the optimizing linkage editor.

7.3.1 Structure of Linkage List

Table 7.5 shows the structure and contents of the linkage list.

Table 7.5 Structure and Contents of Linkage List

No.	Output Information	Contents	When show Option* is Specified	When show Option is not Specified
1	Option information	Option strings specified by a command line or subcommand	None	Output
2	Error information	Error messages	None	Output
3	Linkage map information	Section name, start/end addresses, size, and type	None	Output
4	Symbol information	Static definition symbol name, address, size, and type in the order of address	show =symbol	Not output
		When **show=reference** is specified: Symbol reference count and optimization information in addition to the above information	show =reference	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

Note: * The **show** option is valid when the **list** option is specified.

7.3.2 Option Information

The option strings specified by a command line or a subcommand file are output. Figure 7.5 shows an example of option information output when **optlnk -sub=test.sub -list -show** is specified.

```
(test.sub contents)
INPUT test.obj
```

```
*** Options ***

-sub=test.sub
INPUT test.obj (2)
-list
-show
```
(1)

Figure 7.5 Example of Option Information Output (Linkage List)

(1) Outputs option strings specified by a command line or a subcommand in the specified order.
(2) Subcommand in the **test.sub** subcommand file

7.3.3 Error Information

Error messages are output. Figure 7.6 shows an example of error information output.

```
*** Error Information ***

** L2310 (E) Undefined external symbol “strcmp” referred to in “test.obj”
```
(1)

Figure 7.6 Example of Error Information Output (Linkage List)

(1) Outputs an error message.

7.3.4 Linkage Map Information

The start and end addresses, size, and type of each section are output in the order of address. Figure 7.7 shows an example of linkage map information output.

```
*** Mapping List ***

SECTION                          START      END         SIZE   ALIGN
  (1)                             (2)       (3)          (4)    (5)

P
                                00001000  00001000         1   1
C
                                00001004  00001007         4   4
D_2
                                00001008  000014dd       4d6   2
B_2
                                000014de  000050b3      3bd6   2
```

Figure 7.7 Example of Linkage Map Information Output (Linkage List)

(1) Section name
(2) Start address
(3) End address
(4) Section size
(5) Section boundary alignment value

7.3.5 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. Figure 7.8 shows an example of symbol information output.

```
*** Symbol List ***

SECTION=(1)
FILE=(2)          START           END             SIZE
                   (3)            (4)              (5)
  SYMBOL          ADDR            SIZE            INFO                COUNTS  OPT
  (6)              (7)            (8)              (9)                 (10)   (11)

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
```

Figure 7.8 Example of Symbol Information Output (Linkage List)

(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
(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.

(11) Optimization information as shown below.

ch Symbol modified by optimization

cr Symbol created by optimization

mv Symbol moved by optimization

7.3.6 Symbol Deletion Optimization Information

The size and type of symbols deleted by symbol deletion optimization (**optimize=symbol_delete**) are output. Figure 7.9 shows an example of symbol deletion optimization information output.

```
*** Delete Symbols ***

SYMBOL                          SIZE    INFO
  (1)                           (2)     (3)
  _Version
                                   4    data ,g
```

Figure 7.9 Example of Symbol Deletion Optimization Information Output (Linkage List)

(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

7.3.7 Cross-Reference Information

The symbol reference information (cross-reference information) is output when **show=xreference** is specified. Figure 7.10 shows an example of cross-reference information output.

```
*** Cross Reference List ***

No    Unit Name  Global.Symbol   Location    External Information
(1)     (2)          (3)          (4)              (5)
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
```

Figure 7.10 Example of Cross-Reference Information Output (Linkage List)

(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

(4) Symbol allocation address, which is a relative value from the beginning of the section when **form=rel** is specified

(5) Address of an external symbol that has been referenced

Output format: <Unit number> (<address or offset in section>:<section name>)

7.3.8 Total Section Size

The total sizes of ROM, RAM, and program sections are output. Figure 7.11 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)
```

Figure 7.11 Example of Total Section Size Output (Linkage List)

(1) Total size of RAM data sections

(2) Total size of ROM data sections

(3) Total size of program sections

7.3.9 Vector Information

The contents of the variable vector table are output when **show=vector** is specified. Figure 7.12 shows an example of vector information output.

```
*** Variable Vector Table List ***

NO.     SYMBOL/ADDRESS
(1)     (2)
  0     $fdummy
  1     $fa
  2     00ff8800
  3     $fdummy
    :
  <Omitted>
```

Figure 7.12 Example of Vector Information Output (Linkage List)

(1) Vector number

(2) Symbol. When no symbol is defined for the vector number, the address is output.

7.3.10 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)
```

Figure 7.13 Example of CRC Information Output (Linkage List)

(1) CRC calculation result

(2) Address where the CRC calculation result is output

7.4 Library List

This section covers the contents and format of the library list output by the optimizing linkage editor.

7.4.1 Structure of Library List

Table 7.6 shows the structure and contents of the library list.

Table 7.6 Structure and Contents of Library List

No.	Output Information	Contents	Suboption *	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.

7.4.2 Option Information

The option strings specified by a command line or a subcommand file are output. Figure 7.14 shows an example of option information output when **optlnk –sub = 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
```

(2) (1)

Figure 7.14 Example of Option Information Output (Library List)

(1) Outputs option strings specified by a command line or a subcommand in the specified order.

(2) Subcommand in the **test.sub** subcommand file

7.4.3 Error Information

Messages for errors or warnings are output. Figure 7.15 shows an example of error information output.

```
*** Error Information ***

** L1200 (W) Backed up file “main.lib” into “main.lbk”    (1)
```

Figure 7.15 Example of Error Information Output (Library List)

(1) Outputs a warning message.

7.4.4 Library Information

The library type is output. Figure 7.16 shows an example of library information output.

```
                    *** Library Information  ***

                    LIBRARY NAME=test.lib   (1)
                    CPU=SuperH              (2)
                    ENDIAN=Big              (3)
                    ATTRIBUTE=system        (4)
                    NUMBER OF MODULE=1      (5)
```

Figure 7.16 Example of Library Information Output (Library List)

(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

7.4.5 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.

Figure 7.17 shows an output example of module, section, and symbol information within a library.

```
*** Library List ***

MODULE                LAST UPDATE
 (1)                      (2)
  SECTION
    (3)
    SYMBOL
      (4)
adhry
                      29-Feb-2000 12:34:56
  P
   _main
   _Proc0
   _Proc1
  C
  D
   _Version
  B
   _IntGlob
   _CharGlob
```

Figure 7.17 Example of Module, Section, and Symbol Information Output (Library List)

(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

Section 8 Programming

8.1 Program Structure

8.1.1 Sections

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. If there is even one initial value, the area without initial values is initialized to zero.
- 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.

8.1.2 C/C++ Program Sections

The correspondence between memory areas and sections for C/C++ programs and the standard library is described in table 8.1.

Table 8.1 Summary of Memory Area Types and Their Properties

Name	Section		Format Type	Initial Value	Align-ment Number	Description
	Name	Attribute		Write Operation		
Program area	P*1*6	code	Relative	Yes Not possible	1 byte*7	Stores machine code
Constant area	C*1*2*6*8	romdata	Relative	Yes Not possible	4 bytes	Stores **const** type data
	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	
Initialized data area	D*1*2*6*8	romdata	Relative	Yes Possible	4 bytes	Stores data with initial values
	D_2*1*2*6*8	romdata	Relative	Yes Possible	2 bytes	
	D_1*1*2*6*8	romdata	Relative	Yes Possible	1 byte	
Uninitialized data area	B*1*2*6*8	data	Relative	No Possible	4 bytes	Stores data without initial values
	B_2*1*2*6*8	data	Relative	No Possible	2 bytes	
	B_1*1*2*6*8	data	Relative	No Possible	1 byte	
switch statement 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	
C++ initial processing/ postprocessing data area	C$INIT	romdata	Relative	Yes Not possible	4 bytes	Stores addresses of constructors and destructors called for global class objects
C++ virtual function table area	C$VTBL	romdata	Relative	Yes Not possible	4 bytes	Stores data for calling the virtual function when a virtual function exists in the class declaration
User stack area	SU	data	Relative	No Possible	4 bytes	Area necessary for program execution
Interrupt stack area	SI	data	Relative	No Possible	4 bytes	Area necessary for program execution

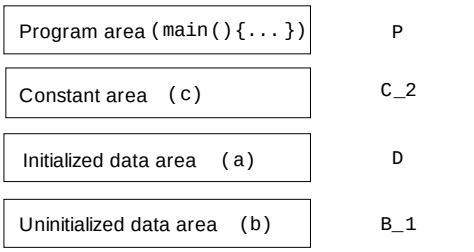
Name	Section		Format Type	Initial Value	Align-ment Number	Description
	Name	Attribute		Write Operation		
Heap area	—	—	Relative	No Possible	—	Area used by library functions **malloc**, **realloc**, **calloc**, and **new**
Absolute address variable area	$ADDR_ <section>_ <address> *[3]	data	Absolute	Yes/No Possible/ Not possible*[4]	—	Stores variables specified by **#pragma address**
Variable vector area	C$VECT	romdata	Relative	No Possible	4 bytes	Variable vector table
Literal area	L*[5]	data	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.
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 or 2.
3. **<section>** is a **C**, **D**, or **B** section name, and **<address>** is an absolute address (hexadecimal).
4. The initial value and write operation depend on the attribute of **<section>**.
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.
6. The section name can be changed through **#pragma section**.
7. Specifying the **instalign4** or **instalign8** option, #**pragma instalign4,** or #**pragma instalign8** changes the boundary alignment to 4 or 8.
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**.

Example 1: A program example is used to demonstrate the correspondence between a C program and the compiler-generated sections.

```
int a=1;
char b;
const short c=0;
void main (){
  ...
}
```

C program



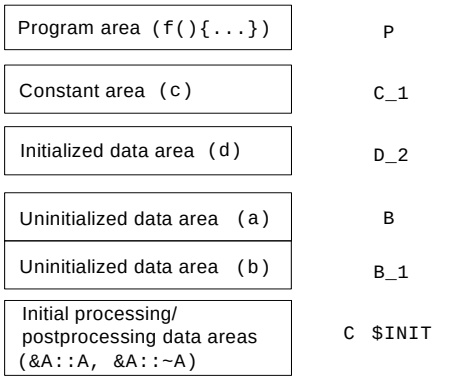
Areas generated by the compiler and stored data

Section name

Example 2: A program example is used to demonstrate the correspondence between a C++ program and the compiler-generated sections.

```
class A{
  int m;
  A(int p);
  ~A();
};
A a(1);
char b;
extern const char c='a';
short d=1;
void f(){...}
```

C++ program



Areas generated by the compiler and stored data

Section name

8.1.3 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 10.3, Assembler Directive Coding.

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.

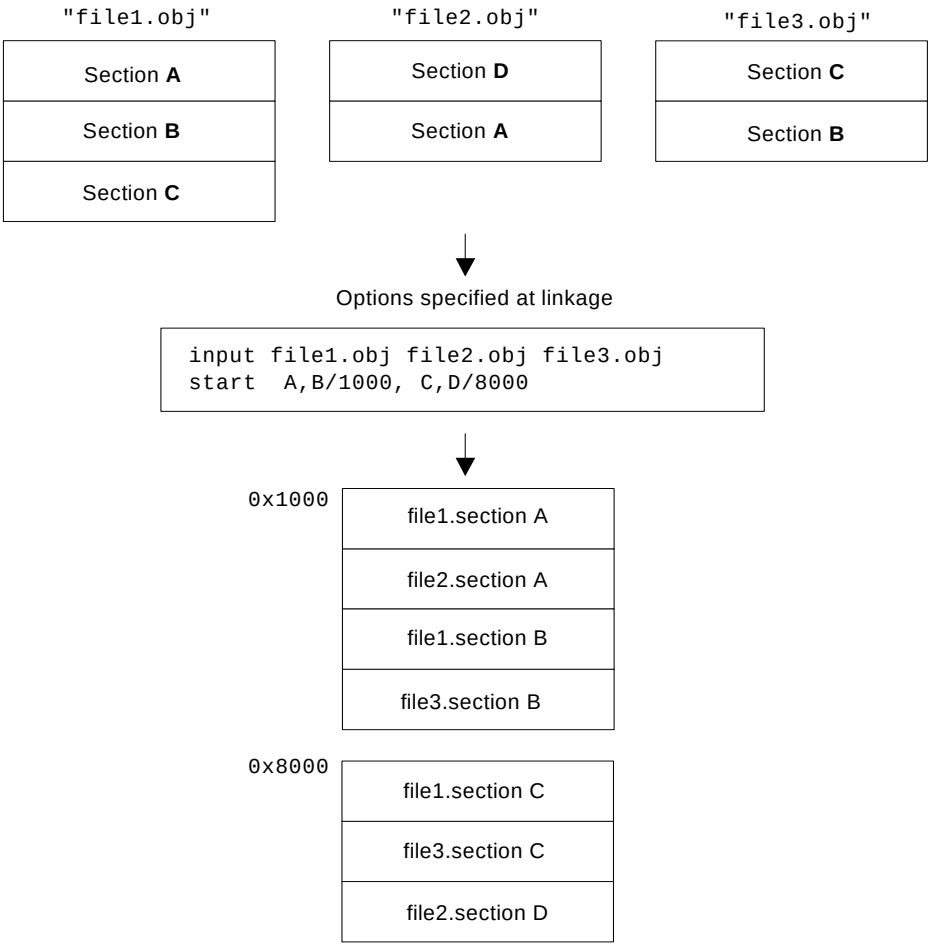
(2) Declares a **romdata** section with section name **B**, allocated address 2000H, and absolute address format.

(3) Declares a **stack** section with section name **C**, boundary alignment 4, and relative address format.

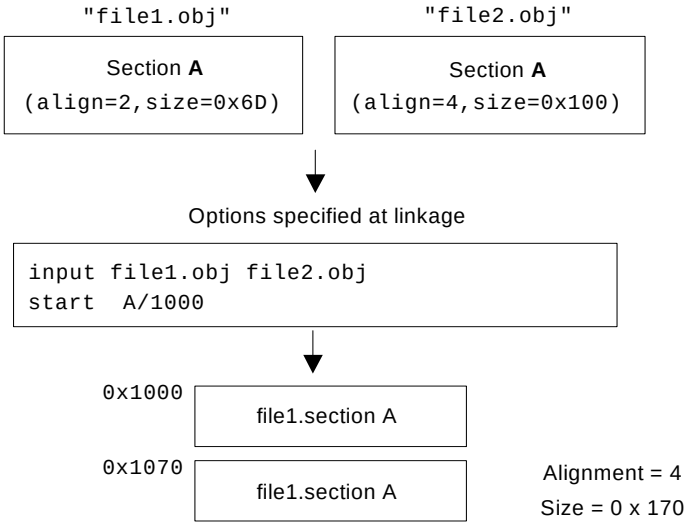
8.1.4 Linking Sections

The optimizing linkage editor links the same sections within input relocatable files, and allocates addresses specified by the **start** option.

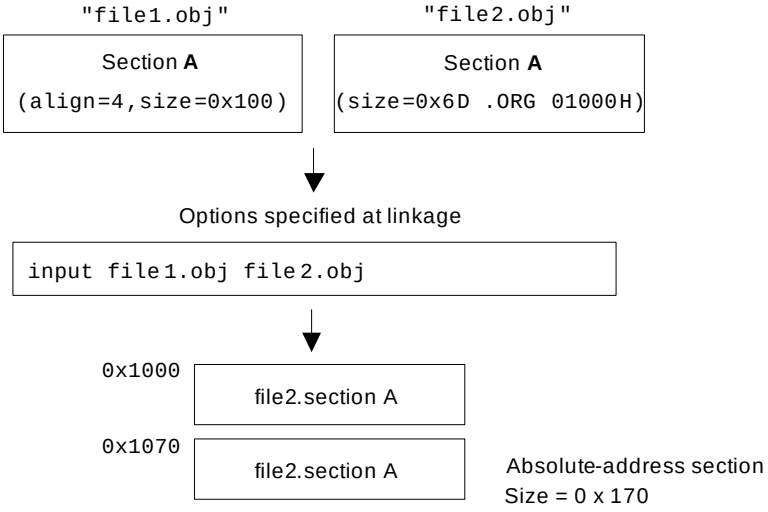
(1) The same section names in different files are allocated continuously in the order of file input.



(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.



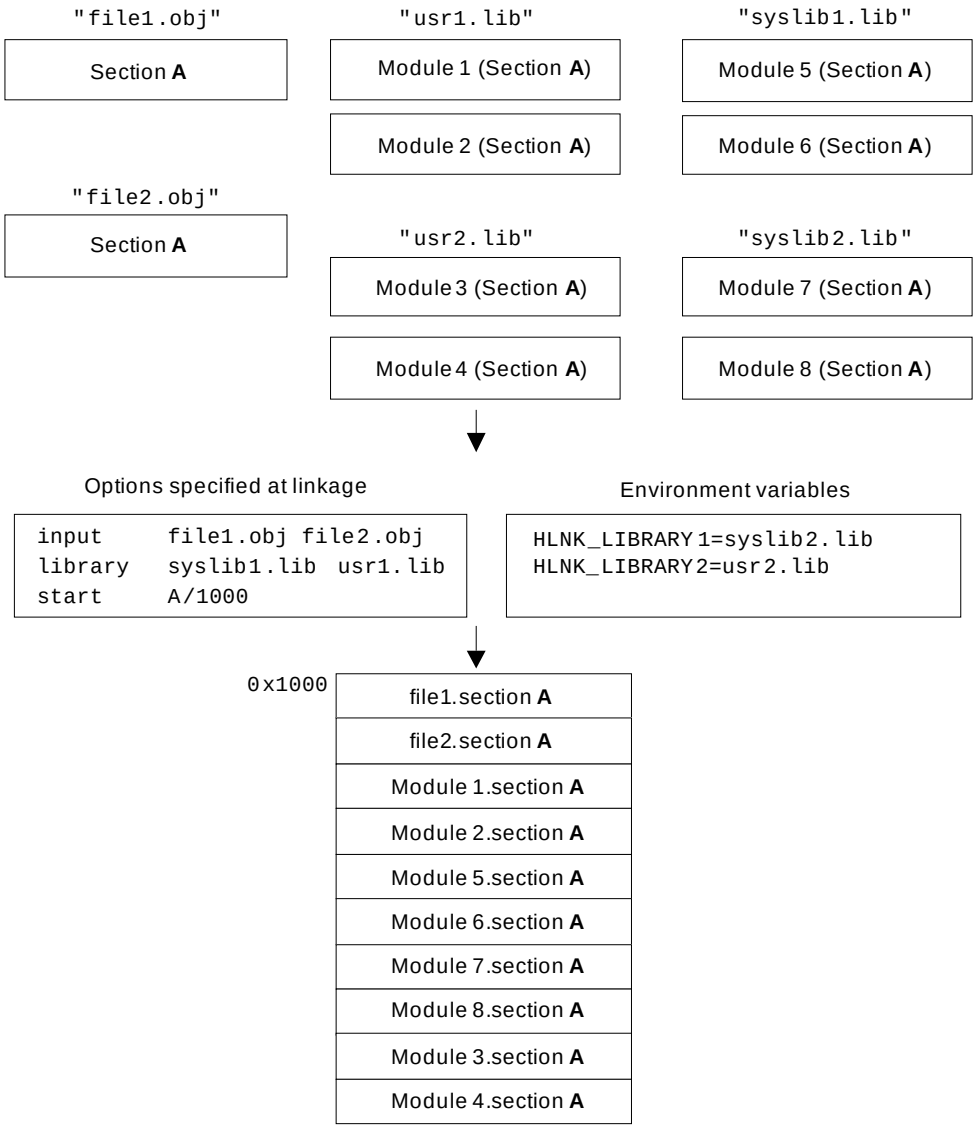
(4) Rules for the order of linking sections with the same name are based on their priorities as follows.

— 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


"file1.obj"
Section A
"usr1.lib"
Module 1 (Section A)
Module 2 (Section A)
"syslib1.lib"
Module 5 (Section A)
Module 6 (Section A)
"file2.obj"
Section A
"usr2.lib"
Module 3 (Section A)
Module 4 (Section A)
"syslib2.lib"
Module 7 (Section A)
Module 8 (Section A)
Options specified at linkage
input file1.obj file2.obj
library syslib1.lib usr1.lib
start A/1000
Environment variables
HLNK_LIBRARY1=syslib2.lib
HLNK_LIBRARY2=usr2.lib
0x1000
file1.section A
file2.section A
Module 1.section A
Module 2.section A
Module 5.section A
Module 6.section A
Module 7.section A
Module 8.section A
Module 3.section A
Module 4.section A

8.2 Function Calling Interface

The rules for using registers and the stack area of the compiler when calling a function are described here. When either a C/C++ program or an assembly program calls the other, the assembly programs must be written using these rules.

- Rules concerning the stack
- Rules concerning registers
- Rules concerning setting and referencing parameters
- Rules concerning setting and referencing return values
- Method for mutual referencing of external names

8.2.1 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 8.1 illustrates the stack frame status immediately after a function call.

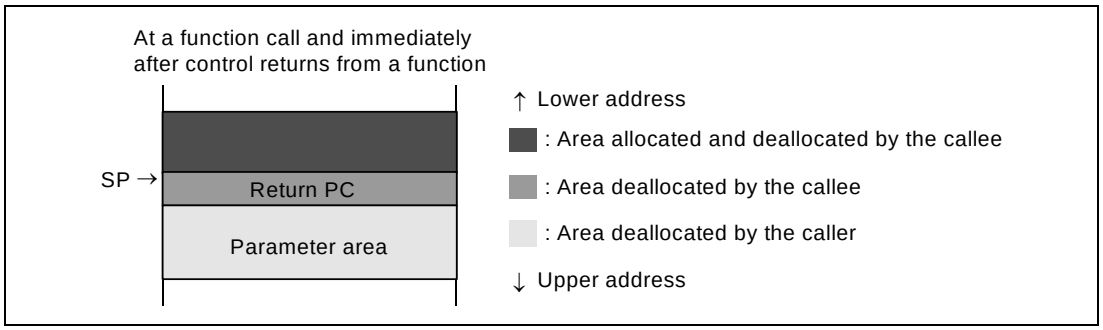


Figure 8.1 Allocation and Deallocation of a Stack Frame

8.2.2 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 8.2 shows the rules for using registers.

Table 8.2 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	—	—	—
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)	—	—	—
ISP USP	Same as R0 when used as the stack pointer. In other cases, the values do not change. *[4]			—	—	—
PC	—	Program counter*[5]		—	—	—

Register	Register Value Does Not Change During Function Call	Function Entry	Function Exit	High-Speed Interrupt Register*1	Base Register*2	PID Register*3
PSW	Not guaranteed	—	(Undefined)	—	—	—
FPSW	Not guaranteed	—	(Undefined)	—	—	—
ACC	Not guaranteed*6	—	(Undefined) *6	—	—	—
INTB	—	No change*4	—	—	—	—
BPC						
BPSW						
FINTV						
CPEN						

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.
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.
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.
4. This does not apply in the case when the registers are set or modified through an intrinsic function or **#pragma inline_asm**.
5. This depends on the specifications of the instruction used for function calls. To call a function, use BSR, JSR, BRA, or JMP.
6. For the instructions that modify the ACC (accumulator), refer to the software manual for the target RX series product.

8.2.3 Rules Concerning Setting and Referencing Parameters

General rules concerning parameters and the method for allocating parameters are described.

Refer to section 8.2.5, 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

- Parameters whose types are declared by a prototype declaration are converted to the declared types.
- 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 8.2 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 8.3 lists general rules on parameter area allocation.

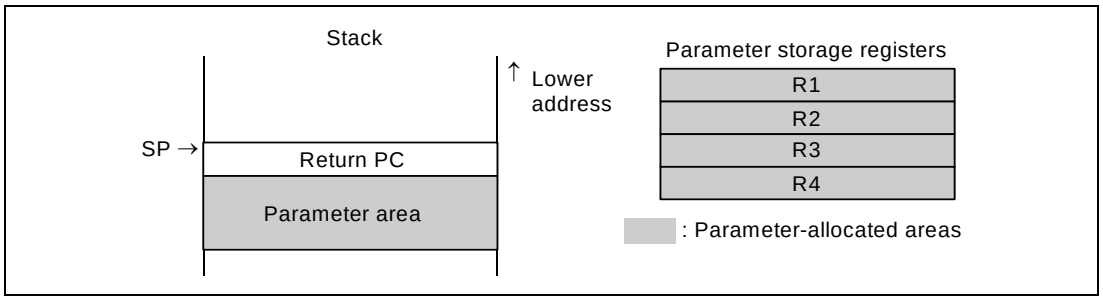


Figure 8.2 Parameter Area Allocation

Table 8.3 General Rules on Parameter Area Allocation

Parameters Allocated to Registers			
Target Type	**Parameter Storage Registers**	**Allocation Method**	**Parameters Allocated to Stack**
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.
2. When **dbl_size=8** is specified.
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 8.3 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**.

8.2.4 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 8.4 for the relationship between the type and the setting area of the return value.

Table 8.4 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.
2. When **dbl_size=8** is not specified.
3. When **dbl_size=8** is specified.

8.2.5 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).

Example 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

Example 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	e	f[1]	f[0]	g

<Stack>

*(R0+0)	s.c[1]	s.c[0]	s.d	
*(R0+4)	Empty area		s.c[3]	s.c[2]

Example 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(C, 1.C, 2, 3, 4)
```

<Registers>

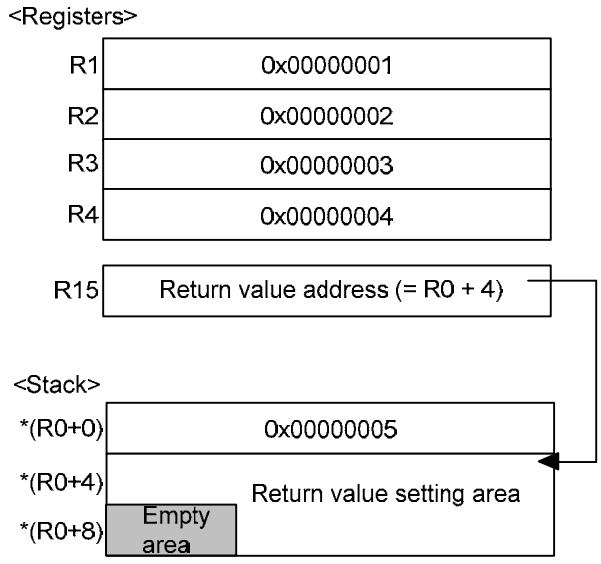
R1	0x00000000
R2	0x3F800000
R3	0x00000002

<Stack>

*(R0+0)	0x00000003
*(R0+4)	0x00000004

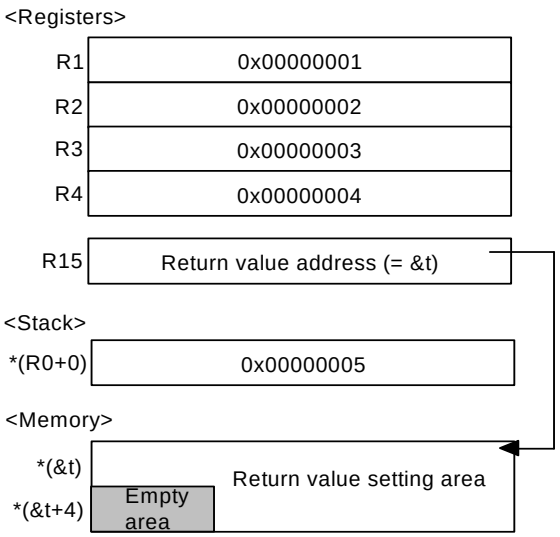
Example 4: When the type returned by a function is more than 16 bytes, or for a structure or union that is not the size of a multiple of 4, the return value address is set to R15.

```
struct S{char a[7];};
struct S f(
   int a1,
   int a2,
   int a3,
   int a4,
   int a5);
   :
f(1,2,3,4,5);
```



Example 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);
```



8.2.6 Method for Mutual Referencing of External Names

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 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 declared as static memory classes (C programs)
- Non-member, non-inline function names not specified as static memory classes (C++ programs)
- Non-inline member function names (C++ programs)
- Static data member names (C++ programs)

(1) Method for referencing assembly program external names in C/C++ programs

In assembly 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.

```
Assembly program (definition)
      . .glb     _a, _b
      .SECTION D,ROMDATA,ALIGN=4
 _a: .LWORD 1
 _b: .LWORD 1
      .END
```

```
C/C++ program (reference)
extern int a,b;

void f()
{
  a+=b;
}
```

(2) Method for referencing C/C++ program external names (variables and C functions) from assembly programs

A C/C++ program can define external variable names (without an underscore (_)).

In an assembly program, .IMPORT is used to declare an external name (preceded by an underscore).

```
C/C++ program (definition)
int a;
```

```
Assembly program (reference)
  .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
```

(3) Method for referencing C++ program external names (functions) from assembly programs

By declaring functions to be referenced from an assembly 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.

C++ program (callee)

```
extern "C"
void sub ( )
{
    :
}
```

Assembly program (caller)

```
.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
```

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 8.4.7, 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**) 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**) at a power-on reset, set the address of **PowerON_Reset** 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(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,
};
```

8.3.2 Initial Setting

The initial setting routine (**PowerON_Reset**) 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** 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** 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** 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** 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, and BPSW are also initialized as required. These registers can be initialized using the embedded functions of the compiler.

Note however that only PSW is not initialized because it holds the interrupt mask setting.

(5) 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")};
```

(6) 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**".

2. In the case of a console or other interactive device, a flag is set to prevent the use of buffering.

(7) 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.

(8) Initialization of PSW for main Function Execution

The PSW register is initialized. The interrupt mask setting is canceled here.

(9) 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 section 9.2.2, Intrinsic Functions.

(10) User Program Execution

The **main** function is executed.

(11) 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_Rese**t function described in section 8.3.2 is shown here.

For the actual initial setting routine created in the integrated development environment, refer to section 8.3.6, Startup Program Created in Integrated Development Environment.

```
#include  <machine.h>
#include  <_h_c_lib.h>
#include  "typedefine.h"
#include  "stacksct.h"

#ifdef __cplusplus
extern "C" {
#endif
void PowerON_Reset(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_init 0x00000100

#pragma section ResetPRG
#pragma entry PowerON_Reset
void PowerON_Reset(void)
```

```
{
    set_intb(__sectop("C$VECT"));
    set_fpsw(FPSW_init);

    _INITSCT();
    _INITLIB();
     nop();
    set_psw(PSW_init);
    main();
    brk();
}
```

8.3.4 Low-Level Interface Routines

When using standard I/O or memory management library functions in a C/C++ program, low-level interface routines must be prepared. Table 8.5 lists the low-level interface routines used by C library functions.

Table 8.5 List of Low-Level Interface Routines

Name	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
error_addr*	Acquires **errno** address
wait_sem*	Defines semaphore
signal_sem*	Releases semaphore

Note: * 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.

Note: 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.

(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.

[Legend]

(Routine name) Simple explanation

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) File Open

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.

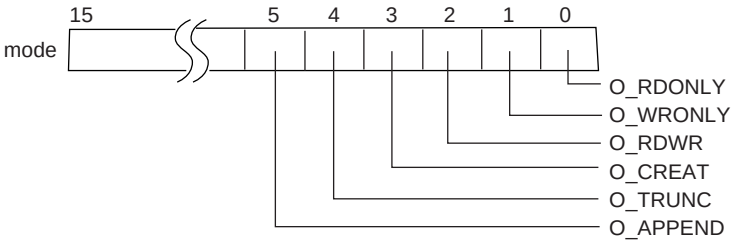


Table 8.6 Explanation of Bits in Parameter "mode" of open Routine

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	Filename 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) File Close

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
Parameter:	fileno	File number of the file to be closed

long read (long fileno, unsigned char *buf, long count) Data Read

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	File 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) Data Write

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	File number 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) File Internal Position Setting

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) Memory Area Allocation

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.

Return value: Normal: Start address of allocated memory
Error: (**char** *) –1

Parameter: 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) Semaphore Allocation

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

Parameter: semnum Semaphore ID

long signal_sem (long semnum) Semaphore Release

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

Parameter: semnum Semaphore ID

(3) Example of Coding Low-Level Interface Routines

```
/************************************************************************/
/*                              lowsrc.c:                               */
/*----------------------------------------------------------------------*/
/*      RX Family Simulator/Debugger Interface Routine                  */
/*              - Only standard I/O (stdin,stdout,stderr) are supported -    */
/************************************************************************/

#include <string.h>
#include <stdio.h>
#include <stddef.h>
#include "lowsrc.h"

/* File Number */
#define STDIN  0                    /* Standard input (Console)        */
#define STDOUT 1                    /* Standard output (Console)       */
#define STDERR 2                    /* Standard error output (Console) */

#define FLMIN  0                    /* Minimum file number             */
#define _MOPENR                0x1
#define _MOPENW                0x2
#define _MOPENA                0x4
#define _MTRUNC                0x8
#define _MCREAT                0x10
#define _MBIN                  0x20
#define _MEXCL                 0x40
#define _MALBUF                0x40
#define _MALFIL                0x80
#define _MEOF                  0x100
#define _MERR                  0x200
#define _MLBF                  0x400
#define _MNBF                  0x800
#define _MREAD                 0x1000
#define _MWRITE                0x2000
#define _MBYTE                 0x4000
#define _MWIDE                 0x8000
/* File flags */
#define O_RDONLY 0x0001 /* Opens in read-only mode.                          */
#define O_WRONLY 0x0002 /* Opens in write-only mode.                         */
#define O_RDWR   0x0004 /* Opens in read/write mode.                         */
#define O_CREAT  0x0008 /* Creates a file if specified file does not exist. */
#define O_TRUNC  0x0010 /* Sets the file size to 0                           */
                        /* when specified file exists.                       */
#define O_APPEND 0x0020 /* Sets the position within the file                 */
                        /* for the next read/write operation.                */
```

```
                        /* 0: Beginning of file  1: End of file.            */
/* Special character code */
#define CR 0x0d                     /* Carriage return                     */
#define LF 0x0a                     /* Line feed                           */

const int _nfiles = IOSTREAM;  /* Specifies the number of input/output files.*/
char flmod[IOSTREAM];          /* Mode setting location of open file         */

unsigned char sml_buf[IOSTREAM];

#define FPATH_STDIN     "C:\\stdin"
#define FPATH_STDOUT    "C:\\stdout"
#define FPATH_STDERR    "C:\\stderr"

/* One character input from standard input */
extern void charput(char);
/* One character output to standard output */
extern char charget(void);
/* One character output to file */
extern char fcharput(char, unsigned char);
/* One character input from file */
extern char fcharget(char*, unsigned char);
/* File open */
extern char fileopen(char*, unsigned char, unsigned char*);
/* File close */
extern char fileclose(unsigned char);
/* File pointer move */
extern char fpseek(unsigned char, long, unsigned char);
/* File pointer get */
extern char fptell(unsigned char, long*);

#include <stdio.h>
FILE *_Files[IOSTREAM]; // File structure
char *env_list[] = { // Environment variable string array (**environ)
    "ENV1=temp01",
    "ENV2=temp02",
    "ENV9=end",
    '\0'            // Environment variable array end NULL
};

char **environ = env_list;

/************************************************************************/
/* _INIT_IOLIB                                                          */
/*  Initialize C library Functions, if necessary.                       */
/*  Define USES_SIMIO on Assembler Option.                              */
/************************************************************************/
void _INIT_IOLIB( void )
{
    /* Opens or creates standard I/O files. Each FILE structure is      */
```

```
    /* initialized in the library. The buffer end pointer reset through     */
    /* freopen() is specified in the _Buf member in each file structure     */
    /* again.                                                               */
    /* Standard input file                                                  */
    if( freopen( FPATH_STDIN, "r", stdin ) == NULL )
        stdin->_Mode = 0xffff;  /* Prohibits access if open processing fails*/
    stdin->_Mode  = _MOPENR;           /* Sets the file for read only.      */
    stdin->_Mode |= _MNBF;             /* Specifies no data buffering.      */
    stdin->_Bend = stdin->_Buf + 1;    /* Sets the buffer end pointer again.*/

    /* Standard output file                                                 */
    if( freopen( FPATH_STDOUT, "w", stdout ) == NULL )
        stdout->_Mode = 0xffff; /* Prohibits access if open processing fails*/
    stdout->_Mode |= _MNBF;            /* Specifies no data buffering.      */
    stdout->_Bend = stdout->_Buf + 1;  /* Sets the buffer end pointer again.*/

    /* Standard error output file                                           */
    if( freopen( FPATH_STDERR, "w", stderr ) == NULL )
        stderr->_Mode = 0xffff; /* Prohibits access if open processing fails*/
    stderr->_Mode |= _MNBF;            /* Specifies no data buffering.      */
    stderr->_Bend = stderr->_Buf + 1;  /* Sets the buffer end pointer again.*/
}

/************************************************************************/
/* _CLOSEALL                                                            */
/************************************************************************/
void _CLOSEALL( void )
{
    int i;

    for( i=0; i < _nfiles; i++ )
    {
        /* Checks if the file is open.                          */
        if( _Files[i]->_Mode & (_MOPENR | _MOPENW | _MOPENA ) )
        fclose( _Files[i] );    /* Closes the file.             */
    }
}

/************************************************************************/
/*       open:file open                                                 */
/*          Return value:File number (Pass)                             */
/*                       -1          (Failure)                          */
/************************************************************************/
long open(const char *name,             /* File name                    */
     long  mode,                        /* Open mode                    */
     long  flg)                         /* Open flag (not used)         */
{


    if( strcmp( name, FPATH_STDIN ) == 0 )   /* Standard input file     */
    {
```

```
        if( ( mode & O_RDONLY ) == 0 ) return -1;
        flmod[STDIN] = mode;
        return STDIN;
    }
    else if( strcmp( name, FPATH_STDOUT ) == 0 )/* Standard output file    */
    {
        if( ( mode & O_WRONLY ) == 0 ) return -1;
        flmod[STDOUT] = mode;
        return STDOUT;
    }
    else if(strcmp(name, FPATH_STDERR ) == 0 ) /* Standard error output file*/
    {
        if( ( mode & O_WRONLY ) == 0 ) return -1;
        flmod[STDERR] = mode;
        return STDERR;
    }
    else return -1;                 /* Files other than standard I/O files */
}

long close( long fileno )
{
    return 1;
}

/************************************************************************/
/* write:Data write                                                     */
/*  Return value:Number of write characters (Pass)                      */
/*               -1                         (Failure)                   */
/************************************************************************/
long  write(long   fileno,             /* File number                    */
      const unsigned char *buf,     /* Transfer destination buffer address */
      long count)                  /* Written character count           */
{
    unsigned long    i;            /* Variable for counting             */
    unsigned char    c;            /* Output character                  */

    /* Checks file mode and outputs one character at a time.            */
    /* Checks if the file is opened in read-only or read/write mode.    */
    if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR)
    {
        if( fileno == STDIN ) return -1;            /* Standard input       */
        else if( (fileno == STDOUT) || (fileno == STDERR)) /*Standard output*/
        {
            for( i = count; i > 0; --i )
            {
                c = *buf++;
                charput(c);
            }
            return count;      /* Returns the number of written characters. */
        }
        else return -1;                 /* Output to file                  */
```

```
    }
    else return -1;                     /* Error                          */
}

long  read( long  fileno, unsigned char *buf, long  count )
{
          unsigned long  i;
       /* Checks mode according to file number, inputs one character each,  */
       /* and stores the characters in buffer.                              */

       if((flmod[fileno]&_MOPENR) || (flmod[fileno]&O_RDWR)){
             for(i = count; i > 0u; i--){
                   *buf = charget();
                   if(*buf==CR){           /* Replaces line feed character. */
                         *buf = LF;
                   }
                   buf++;
             }
             return count;
       }
       else {
             return -1;
       }
}

long lseek( long fileno, long offset, long base )
{
    return -1L;
}

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          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 a low-level interface routine for a reentrant library. This routine is necessary when using a standard library, which was created by the standard library generator with the **reent** option specified.

When an error is returned from the **wait_sem** function or **signal_sem** function, set **errno** as follows to return from the library function.

Bit Function	errno	Description
wait_sem	EMALRESM	Failed to allocate semaphore resources for **malloc**
	ETOKRESM	Failed to allocate semaphore resources for **strtok**
	EIOBRESM	Failed to allocate semaphore resources for **_iob**
signal_sem	EMALFRSM	Failed to release semaphore resources for **malloc**
	ETOKFRSM	Failed to release semaphore resources for **strtok**
	EIOBFRSM	Failed to release semaphore resources for **_iob**

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 _iob   */
#define SEMSIZE           4
#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);

long sem_errno;
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** 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
}
```

8.3.6 Startup Program Created in Integrated Development Environment

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 8.7.

Table 8.7 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
(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)

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(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_init 0x00000000        // FPSW 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 to PResetPRG section

#pragma entry PowerON_Reset

void PowerON_Reset(void)
{
    set_intb(__sectop("C$VECT"));

#ifdef __ROZ                // Initialize FPSW
#define _ROUND 0x00000001   // Let FPSW RMbits=01 (round to zero)
#else
#define _ROUND 0x00000000   // Let FPSW RMbits=00 (round to nearest)
#endif
#ifdef __DOFF
#define _DENOM 0x00000100   // Let FPSW DNbit=1 (denormal as zero)
#else
#define _DENOM 0x00000000   // Let FPSW DNbit=0 (denormal as is)
#endif
```

```
    set_fpsw(FPSW_init | _ROUND | _DENOM);

    _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
    nop();

//  _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,
```

```
//;0xfffffffe0  Reserved
    Dummy,
//;0xfffffffe4  Exception (Floating Point)
    Excep_FloatingPoint,
//;0xfffffffe8  Reserved
    Dummy,
//;0xfffffffec  Reserved
    Dummy,
//;0xffffffff0  Reserved
    Dummy,
//;0xffffffff4  Reserved
    Dummy,
//;0xffffffff8  NMI
    NonMaskableInterrupt,
//;0xffffffffc  RESET
//;<<VECTOR DATA START (POWER ON RESET)>>
//;Power On Reset PC
PowerON_Reset
//;<<VECTOR DATA END (POWER ON RESET)>>
};
```

(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 L1100 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 _MOPENR    0x1
#define _MOPENW    0x2
#define _MOPENA    0x4
#define _MTRUNC    0x8
#define _MCREAT    0x10
#define _MBIN      0x20
#define _MEXCL     0x40
#define _MALBUF    0x40
#define _MALFIL    0x80
#define _MEOF      0x100
#define _MERR      0x200
#define _MLBF      0x400
#define _MNBF      0x800
#define _MREAD     0x1000
#define _MWRITE    0x2000
#define _MBYTE     0x4000
#define _MWIDE     0x8000

#define O_RDONLY   0x0001
#define O_WRONLY   0x0002
#define O_RDWR     0x0004
#define O_CREAT    0x0008
#define O_TRUNC    0x0010
#define O_APPEND   0x0020

#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 )
{
    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(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 -cpu=rx600 -output=LoadModule.lib
ccrx -cpu=rx600 -output=obj UserProgram.c
ccrx -cpu=rx600 -output=obj resetprg.c
ccrx -cpu=rx600 -output=obj intprg.c
ccrx -cpu=rx600 -output=obj vecttbl.c
ccrx -cpu=rx600 -output=obj dbsct.c
ccrx -cpu=rx600 -output=obj lowsrc.c
asrx -cpu=rx600 lowlvl.src
ccrx -cpu=rx600 -output=obj sbrk.c
optlnk -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
optlnk -output=LoadModule.sty -form=stype -output=LoadModule.mot LoadModule.abs
```

8.4 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.

8.4.1 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.

8.4.2 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 in section 2 (compiler), section 4 (assembler), and section 5 (optimizing linkage editor).

(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**, **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.

8.4.3 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 section 8.4.7, Application Startup.

8.4.4 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.

8.4.5 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 8.8.

Table 8.8 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	**nouse_pid_register** must be 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

(2) Application

When building an application, specify the PIC/PID function options as shown in table 8.9.

Table 8.9 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 8.10.

Table 8.10 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
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.

8.4.6 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 8.3, 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,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 */
```

8.4.7 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 8.3.2 (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 8.3.2 (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 8.3.2 (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 8.3.2 (6).

(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 8.3.2 (8) and 8.3.2 (9).

(7) User Program Execution

Execute the **main** function.

Specify the processing by referring to section 8.3.2 (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 8.3.2 (5) to support the PID function.
  Since the PID register is fixed at R13 in this example, change R13 to the desired register when another register is used as the PID register.
- **initiolib.c**; Contains **_INITLIB**, which initializes the standard libraries.
  This is created by modifying the code described in section 8.3.2 (6) to be used for the application.

[startup_picpid.c]

```
// Initialization Processing Described in Section 8.3.2(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 8.3.2 (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,R13,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
}
```

Section 9 C/C++ Language Specifications

9.1 Language Specifications

9.1.1 Compiler Specifications

The following shows compiler specifications for the implementation-defined items which are not prescribed in the language specifications.

For the language specifications that this compiler conforms to, refer to section 9.1.5, Conforming Language Specifications

(1) Environment

Table 9.1 Environment Specifications

No.	Item	Compiler Specifications
1	Purpose of actual argument for the **main** function	Not stipulated
2	Structure of interactive I/O devices	Not stipulated

(2) Identifiers

Table 9.2 Identifier Specifications

No.	Item	Compiler Specifications
1	Number of valid letters in non externally-linked identifiers (internal names)	Up to 8189 letters in both external and internal names
2	Number of valid letters in externally-linked identifiers (external names)	Up to 8191 letters in both external and internal names
3	Distinction of uppercase and lowercase letters in externally-linked identifiers (external names)	Uppercase and lowercase letters are distinguished

(3) Characters

Table 9.3 Character Specifications

No.	Item	Compiler Specifications
1	Elements of source character sets and execution environment character sets	Source program character sets and execution environment character sets are both ASCII character sets. However, strings and character constants can be written in shift JIS or EUC Japanese character code, Latin1 code, or UTF-8 code.
2	Shift states used in coding multibyte characters	Shift states are not supported.
3	Number of bits for a character in character sets in program execution	8 bits
4	Relationship between source program character sets in character constants and strings and characters in execution environment character sets	Corresponds to same ASCII characters.
5	Values of integer character constants that include characters or escape sequences which are not stipulated in the language specifications	Characters and escape sequences which are not stipulated in the language specifications are not supported.
6	Values of character constants that include two or more characters, and wide character constants that include two or more multibyte characters	The first two bytes of character constants are valid. Wide character constants are not supported. Note that a warning error message is output if you specify more than one character.
7	Specifications of **locale** used for converting multibyte characters to wide characters	locale is not supported.
8	**char** type value	Same value range as **unsigned char** type*.

Note: * The **char** type has the same value range as the **signed char** type when the **signed_char** option is specified.

(4) Integers

Table 9.4 Integer Specifications

No.	Item	Compiler Specifications
1	Representation and values of integer types	See table 9.5.
2	Values when integers are converted to shorter signed integer types or unsigned integers are converted to signed integer types of the same size (when converted values cannot be represented by the target type)	The least significant four bytes, two bytes, or one byte of the integer value will respectively be the post-conversion value when the size of the post-conversion type is four bytes, two bytes, or one byte.
3	Result of bit-wise operations on signed integers	Signed value.
4	Remainder sign in integer division	Same sign as dividend.
5	Result of right shift of signed scalar types with a negative value	Maintains sign bit.

Table 9.5 Range of Integer Types and Values

No.	Type	Value Range	Data Size
1	char *[1]	0 to 255	1 byte
2	signed char	–128 to 127	1 byte
3	unsigned char	0 to 255	1 byte
4	short, signed short	–32768 to 32767	2 bytes
5	unsigned short	0 to 65535	2 bytes
6	int*[2], signed int*[2]	–2147483648 to 2147483647	4 bytes
7	unsigned int*[2]	0 to 4294967295	4 bytes
8	long, signed long	–2147483648 to 2147483647	4 bytes
9	unsigned long	0 to 4294967295	4 bytes
10	long long, signed long long	–9223372036854775808 to 9223372036854775807	8 bytes
11	unsigned long long	0 to 18446744073709551615	8 bytes

Notes: 1. When the **signed_char** option is specified, the **char** type is handled as the **signed char** type.
2. When the **int_to_short** option is specified, the **int** type is handled as the **short** type, the **signed int** type as the **signed short** type, and the **unsigned int** type as the **unsigned short** type.

(5) Floating-Point Numbers

Table 9.6 Floating-Point Number Specifications

No.	Item	Compiler Specifications
1	Representation and values of floating-point types	There are three types of floating-point numbers: **float**, **double**, and **long double** types. See section 9.1.3, Floating-Point Number Specifications, for the internal representation of floating-point types and specifications for their conversion and operation. Table 9.7 shows the limits of floating-point type values that can be expressed.
2	Method of truncation when integers are converted into floating-point numbers that cannot accurately represent the actual value	
3	Methods of truncation or rounding when floating-point numbers are converted into shorter floating-point types	

Table 9.7 Limits of Floating-Point Type Values

		Limits	
No.	Item	Decimal Notation*[1]	Internal Representation (Hexadecimal)
1	Maximum value of **float** type	3.4028235677973364e+38f (3.4028234663852886e+38f)	7f7fffff
2	Minimum positive value of **float** type	7.0064923216240862e−46f (1.4012984643248171e−45f)	00000001
3	Maximum values of **double** type and **long double** type*[2]	1.7976931348623158e+308 (1.7976931348623157e+308)	7fefffffffffffff
4	Minimum positive values of **double** type and **long double** type *[2]	4.9406564584124655e−324 (4.9406564584124654e−324)	0000000000000001

Notes: 1. The limits for decimal notation are the maximum value smaller than infinity and the minimum value greater than 0. Values in parentheses are theoretical values.
2. These values are the limits when **dbl_size=8** is specified. When **dbl_size=4** is specified, the **double** type and **long double** type have the same value as the **float** type.

(6) Arrays and Pointers

Table 9.8 Array and Pointer Specifications

No.	Item	Compiler Specifications
1	Integer type (**size_t**) required to hold maximum array size	**unsigned long** type
2	Conversion from pointer type to integer type (pointer type size >= integer type size)	Value of least significant byte of pointer type
3	Conversion from pointer type to integer type (pointer type size < integer type size)	Zero extension
4	Conversion from integer type to pointer type (integer type size >= pointer type size)	Value of least significant byte of integer type
5	Conversion from integer type to pointer type (integer type size < pointer type size)	Sign extension
6	Integer type (**ptrdiff_t**) required to hold difference between pointers to members in the same array	**int** type

(7) Registers

Table 9.9 Register Specifications

No.	Item	Compiler Specifications
1	Types of variables that can be assigned to registers	char, signed char, unsigned char, bool, _Bool, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, pointer

(8) Class, Structure, Union, and Enumeration Types, and Bit Fields

Table 9.10 Class, Structure, Union, and Enumeration Types, and Bit Field Specifications

No.	Item	Compiler Specifications
1	Referencing members in union type accessed by members of different types	Can be referenced but value cannot be guaranteed.
2	Boundary alignment of class and structure members	The maximum alignment value of the class and structure members is used as the boundary alignment value. For details on assignment, see section 9.1.2 (2), Compound Type (C), Class Type (C++).
3	Sign of bit fields of simple **int** type	**unsigned int** type *[3]
4	Order of bit fields within **int** type size	Assigned from least significant bit.*[1] *[2]
5	Method of assignment when the size of a bit field assigned after a bit field is assigned within an **int** type size exceeds the remaining size in the **int** type	Assigned to next **int** type area.*[1]
6	Type specifiers allowed for bit fields	char, unsigned char, bool, _Bool, short, unsigned short, int, unsigned int, long, unsigned long, enum, long long, unsigned long long
7	Integer type representing value of enumeration type	**int** type*[4]

Notes:
1. For details of assignment of bit fields, see section 9.1.2 (3), Bit Fields.
2. Specifying the **bit_order=left** option assigns bit fields from the most significant bit.
3. When the **signed_bitfield** option is specified, the sign of bit fields is handled as the **signed int** type.
4. When the **auto_enum** option is specified, the smallest type that holds enumeration values is selected. For details, refer to the description of the **auto_enum** option in section 2.5, Microcontroller Options.

(9) Type Qualifiers

Table 9.11 Type Qualifier Specifications

No.	Item	Compiler Specifications
1	Types of access to data qualified with **volatile**	Not stipulated

(10) Declarations

Table 9.12 Declaration Specifications

No.	Item	Compiler Specifications
1	Number of declarations modifying basic types (arithmetic types, structure types, union types)	16 max.

The following shows examples of counting the number of types modifying basic types.

i. `int a;` Here, a has an **int** type (basic type) and the number of types modifying the basic type is 0.

ii. `char *f();` Here, f has a function type returning a pointer type to a **char** type (basic type), and the number of types modifying the basic type is 2.

(11) Statements

Table 9.13 Statement Specifications

No.	Item	Compiler Specifications
1	Number of **case** labels that can be declared in one **switch** statement	2,147,483,646 max.

(12) Preprocessor

Table 9.14 Preprocessor Specifications

No.	Item	Compiler Specifications
1	Relationship between single-character character constants in constant expressions in a conditional inclusion, and execution environment character sets	Preprocessor statement character constants are the same as the execution environment character set.
2	Method of reading include files	Files enclosed in "<" and ">" are read from the directory specified in the **include** option. If the specified file is not found, the directory specified in environment variable **INC_RX** is searched, followed by the directory specified in environment variable **BIN_RX**.
3	Support for include files enclosed in double-quotes	Supported. Include files are read from the current directory. If not found in the current directory, the file is searched for as described in 2, above.
4	Space characters in strings after a macro is expanded	A string of space characters are expanded as one space character.
5	Operation of **#pragma** statements	See section 9.2.1, #pragma Extension Specifiers and Keywords.
6	_ _**DATE**_ _ and _ _**TIME**_ _ values	Values are specified based on the host computer's timer at the start of compiling.

9.1.2 Internal Data Representation

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 9.15 shows internal representation of scalar type data in C and basic type data in C++.

Table 9.15 Internal Representation of Scalar-Type and Basic-Type Data

No.	Data Type	Size (bytes)	Align-ment (bytes)	Sign	Data Range	
					Minimum Value	Maximum Value
1	char*[1]	1	1	Unused	0	2^8–1 (255)
2	signed char	1	1	Used	-2^7 (–128)	2^7–1 (127)
3	unsigned char	1	1	Unused	0	2^8–1 (255)
4	short	2	2	Used	-2^{15} (–32768)	2^{15}–1 (32767)
5	signed short	2	2	Used	-2^{15} (–32768)	2^{15}–1 (32767)
6	unsigned short	2	2	Unused	0	2^{16}–1 (65535)
7	int*[2]	4	4	Used	-2^{31} (–2147483648)	2^{31}–1 (2147483647)
8	signed int*[2]	4	4	Used	-2^{31} (–2147483648)	2^{31}–1 (2147483647)
9	unsigned int*[2]	4	4	Unused	0	2^{32}–1 (4294967295)
10	long	4	4	Used	-2^{31} (–2147483648)	2^{31}–1 (2147483647)
11	signed long	4	4	Used	-2^{31} (–2147483648)	2^{31}–1 (2147483647)

No.	Data Type	Size (bytes)	Alignment (bytes)	Sign	Data Range	
					Minimum Value	Maximum Value
12	unsigned long	4	4	Unused	0	2^{32}–1 (4294967295)
13	long long	8	4	Used	-2^{63} (–9223372036854775808)	2^{63}–1 (9223372036854775807)
14	signed long, long	8	4	Used	-2^{63} (–9223372036854775808)	2^{63}–1 (9223372036854775807)
15	unsigned long, long	8	4	Unused	0	2^{64}–1 (18446744073709551615)
16	float	4	4	Used	–∞	+∞
17	double, long double	4*4	4	Used	–∞	+∞
18	size_t	4	4	Unused	0	2^{32}–1 (4294967295)
19	ptr_diff_t	4	4	Used	-2^{31} (–2147483648)	2^{31}–1 (2147483647)
20	enum*3	4	4	Used	-2^{31} (–2147483648)	2^{31}–1 (2147483647)
21	Pointer	4	4	Unused	0	2^{32}–1 (4294967295)
22	bool*5 _Bool*8	1	1	—*9	—	—
23	Reference*6	4	4	Unused	0	2^{32}–1 (4294967295)
24	Pointer to a data member*6	4	4	Used	0	2^{32}–1 (4294967295)
25	Pointer to a function member*6 *7	12	4	—*9	—	—

Notes:
1. When the **signed_char** option is specified, the **char** type is the same as the **signed char** type.
2. When the **int_to_short** option is specified, the **int** type is the same as the **short** type, the **signed int** type as the **signed short** type, and the **unsigned int** type as the **unsigned short** type.
3. When the **auto_enum** option is specified, the smallest type that holds enumeration values is selected.
4. When **dbl_size=8** is specified, the size of the **double** type and **long double** type is 8 bytes.
5. This data type is only valid for compilation of C++ programs or C99 programs including **stdbool.h**.
6. These data types are only valid for compilation of C++ programs.
7. Pointers to function and virtual function members are represented in the following data structure.

```
class _PMF{
  public:
    long d;             // Object offset value.
    long i;             // Index in the virtual function table
                        // when the target function is
                        // the virtual function.

    union{
      void (*f)();      // Address of a function when the target function
                        // is a non-virtual function.
      long offset;      // Object offset value of the virtual function table
                        // when the target function is the virtual function.
    };
};
```

8. This data type is only valid for compilation in C99. The **_Bool** type is treated as the **bool** type in compilation.
9. 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 9.16 shows internal representation of compound type and class type data.

Table 9.16 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

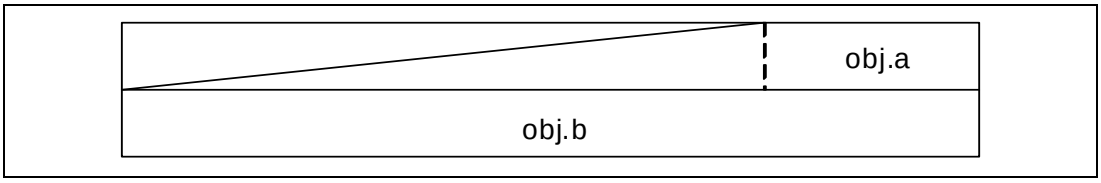
In the following examples, a rectangle (▭) indicates four bytes. The diagonal line (／) represents an unused area for alignment. The address increments from right to left (the left side is located at a higher address).

(a) Structure Data Allocation

When structure members are allocated, an unused area may be generated between structure members to align them to boundaries.

Example:

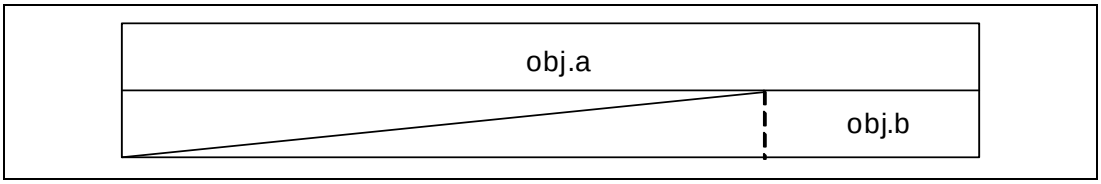
```
struct {
 char a;
 int b;
} obj
```



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
```

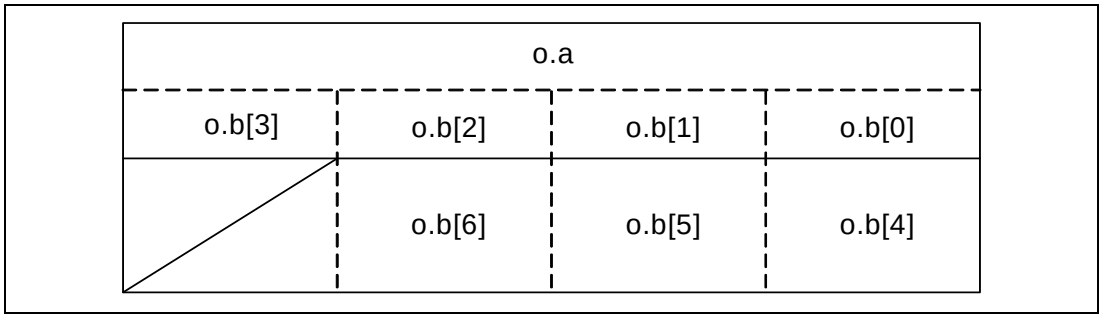


(b) Union Data Allocation

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;
```

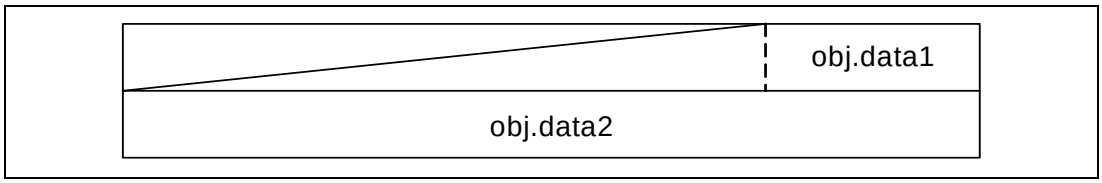


(c) Class Data Allocation

For classes having no base class or virtual functions, data members are allocated according to the allocation rules of structure data.

Example:

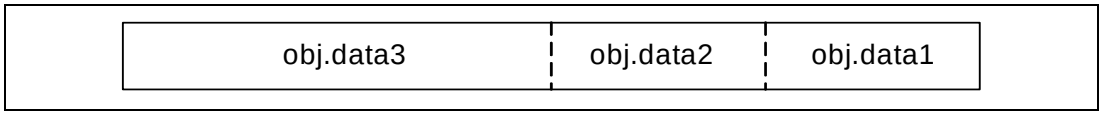
```
class A{
  char data1;
  int data2;
public:
  A();
  int getData1(){return data1;}
}obj;
```



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:

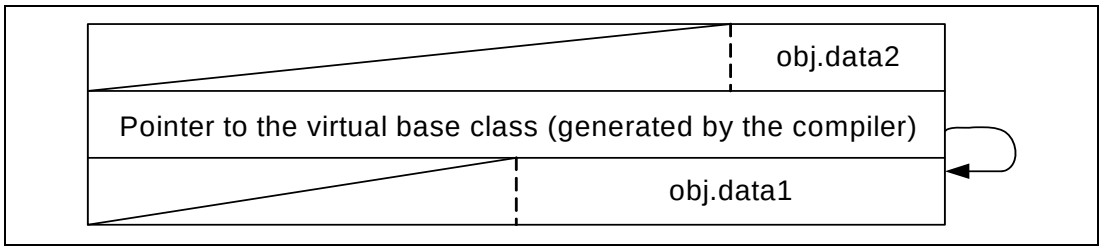
```
class A{
  char data1;
};
class B:public A{
  char data2;
  short data3;
}obj;
```



For a class having a virtual base class, a pointer to the virtual base class is allocated.

Example:

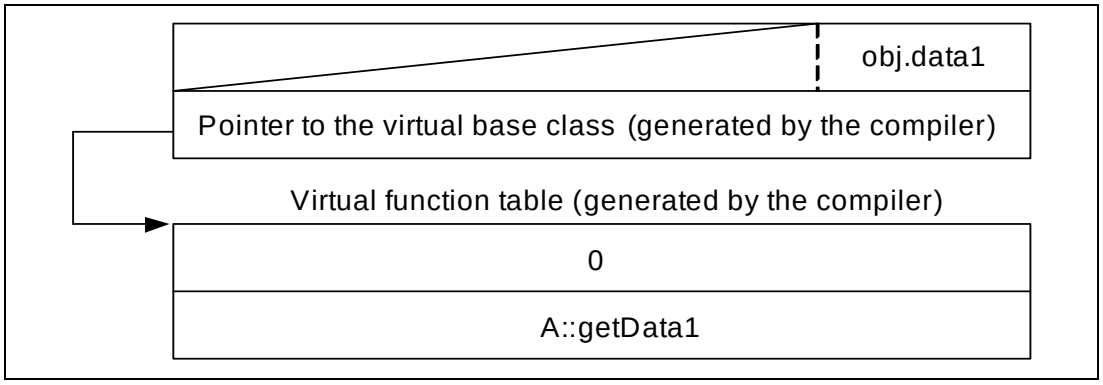
```
class A{
   short data1;
};
class B: virtual protected A{
   char data2;
}obj;
```



For a class having virtual functions, the compiler creates a virtual function table and allocates a pointer to the virtual function table.

Example:

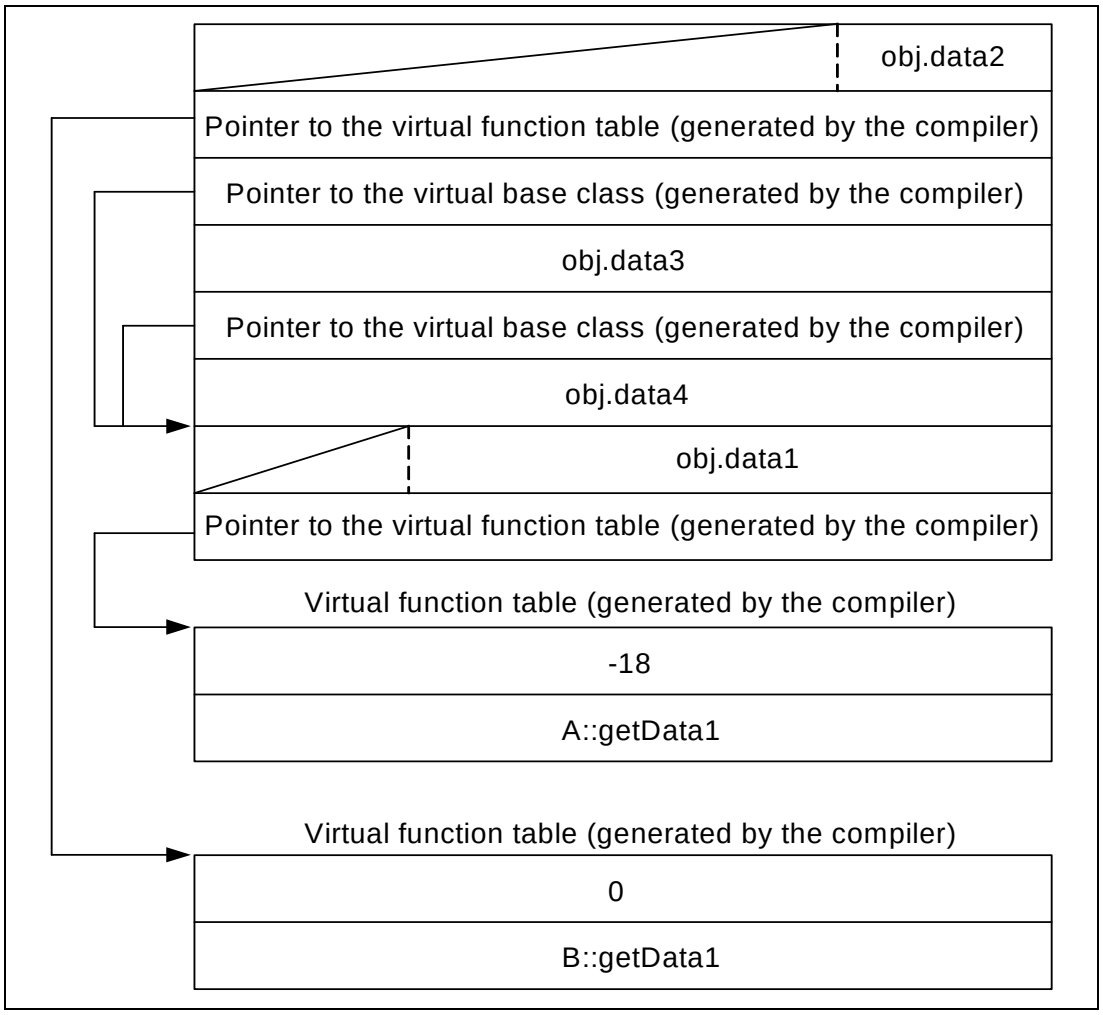
```
class A{
    char data1;
  public:
    virtual int getData1();
}obj;
```



An example is shown for class having virtual base class, base class, and virtual functions.

Example:

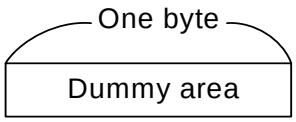
```
class A{
  char data1;
  virtual short getData1();
};
class B:virtual public A{
  char data2;
  char getData2();
  short getData1();
};
class C:virtual protected A{
  int data3;
};
class D:virtual public A,public B,public C{
  public:
  int data4;
  short getData1();
}obj;
```



For an empty class, a 1-byte dummy area is assigned.

Example:

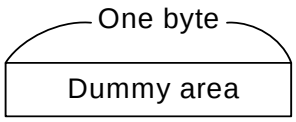
```
class A{
  void fun();
}obj;
```



For an empty class having an empty class as its base class, the dummy area is one byte.

Example:

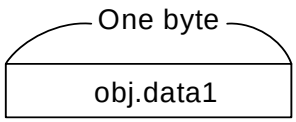
```
class A{
  void fun();
};
class B: A{
  void sub();
}obj;
```



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 9.17 shows the specifications of bit field members.

Table 9.17 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 C99 programs including **stdbool.h**.
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.
3. Zero extension: Zeros are written to the upper bits to extend data.
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.
5. This data type is only valid for programs in C99. The **_Bool** type is treated as the **bool** type in compilation.

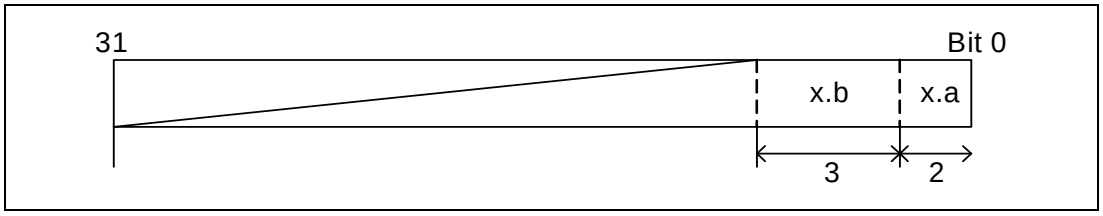
(b) Bit Field Allocation

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:

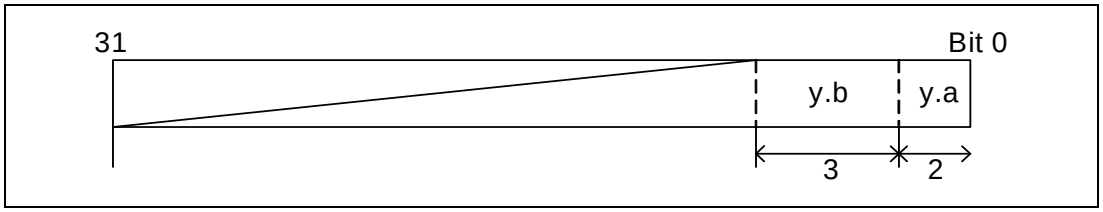
```
struct b1 {
  int a:2;
  int b:3;
} x;
```



- Consecutive bit field members having type specifiers of the same size are placed in the same area as much as possible.

Example:

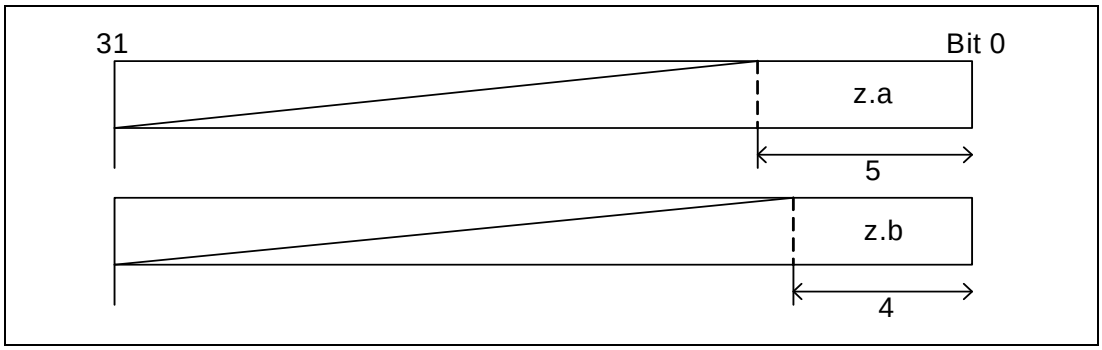
```
struct b1 {
  long          a:2;
  unsigned int  b:3;
} y;
```



- Bit field members having type specifiers with different sizes are allocated to separate areas.

Example:

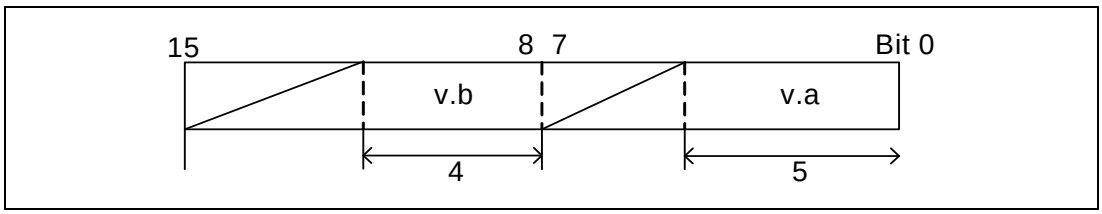
```
struct b1 {
  int    a:5;
  char   b:4;
} z;
```



- 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:

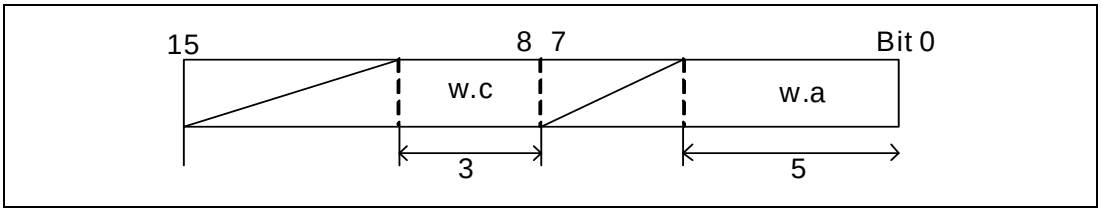
```
struct b2 {
  char   a:5;
  char   b:4;
} v;
```



- 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 section 2, Compiler Options, and the description on **#pragma bit_order** in section 9.2.1, #pragma Extension Specifiers and Keywords.

(4) Memory Allocation in Big Endian

In big endian, data are allocated in the memory as follows:

(a) One-Byte Data ((signed) char, unsigned char, bool, and _Bool types)

The order of bits in one-byte data for the little endian and the big endian is the same.

(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, unsigned int, (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.

Example: When four-byte data 0x12345678 is allocated at address 0x100:

Little Endian:	Address 0x100: 0x78	Big Endian: 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:	Address 0x100: 0xef	Big Endian: 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:	Address 0x100: 0x34	Big Endian: 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:	Address 0x100: 0x01	Big Endian: 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

9.1.3 Floating-Point Number Specifications

(1) 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.

(a) 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.

(b) Structure of Internal Representation

Figure 9.1 shows the structure of the internal representation of **float**, **double**, and **long double** types.

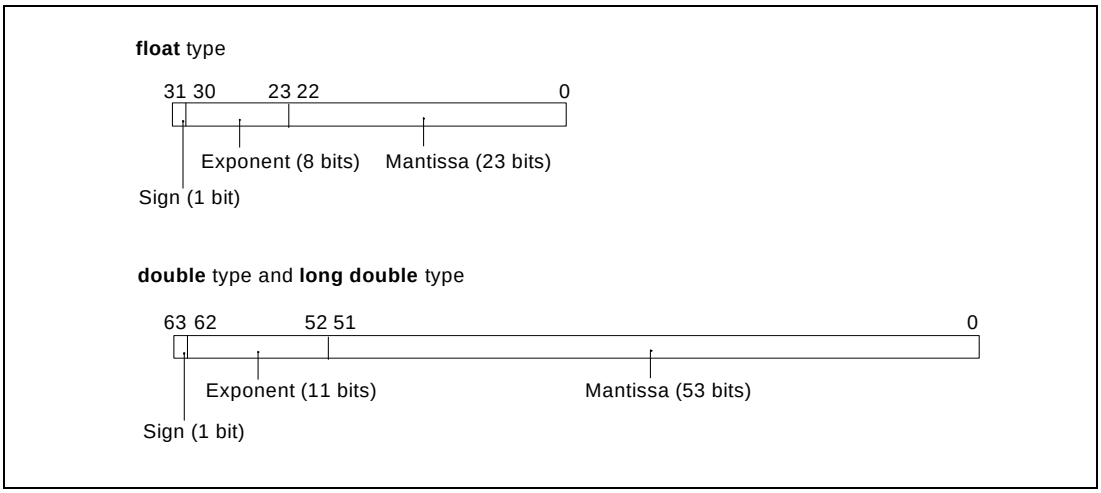


Figure 9.1 Structure of Internal Representation of Floating-Point Numbers

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.

(c) 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 9.18 shows the types of values represented as floating-point numbers.

Table 9.18 Types of Values Represented as Floating-Point Numbers

	Exponent		
Mantissa	**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.

(2) float Type

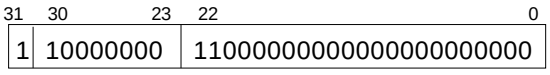
The **float** type is internally represented by a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa.

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:



Sign: –

Exponent: $10000000_{(2)} - 127 = 1$, where $_{(2)}$ indicates binary

Mantissa: $1.11_{(2)} = 1.75$

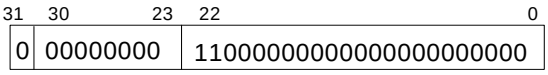
Value: $-1.75 \times 2^1 = -3.5$

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:



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. See section 9.1.3 (4), Floating-Point Operation Specifications, for the functional differences deriving from the sign used with zero.

iv. Infinity

The sign is 0 (positive) or 1 (negative), indicating +∞ or –∞, 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.

(3) double Types and long double Types

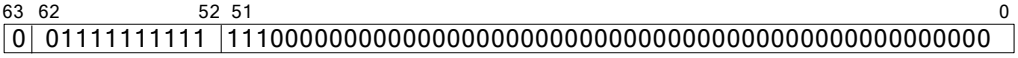
The **double** and **long double** types are internally represented by a 1-bit sign, an 11-bit exponent, and a 52-bit mantissa.

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:



Sign: +

Exponent: $1111111111_{(2)} - 1023 = 0$, where $_{(2)}$ indicates binary

Mantissa: $1.111_{(2)} = 1.875$

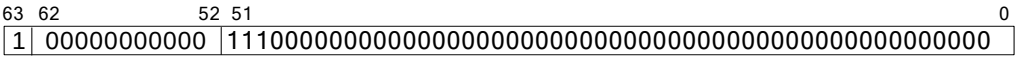
Value: $1.875 \times 2^0 = 1.875$

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:



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. See section 9.1.3 (4), Floating-Point Operation Specifications, for the functional differences deriving from the sign used with zero.

iv. Infinity

The sign is 0 (positive) or 1 (negative), indicating +∞ or –∞, 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) Floating-Point Operation Specifications

This section describes the specifications for arithmetic operations on floating-point numbers in C/C++, and for conversion between the decimal representation of floating-point numbers and their internal representation during compilation and in library processing.

(a) Specifications for Arithmetic Operations

i. Rounding of results

When the result of arithmetic operations on floating-point numbers exceeds the number of valid digits in the mantissa in internal representation, the result is rounded according to the following rules:

a. The result is rounded toward the closer of the two internal representations of the approximating floating-point numbers.
b. When the result is exactly between the two approximating floating-point numbers, it is rounded to the floating-point number of which the last digit of the mantissa is 0.

ii. Processing of overflows, underflows, and illegal operations

The following is performed in the event of an overflow, underflow, or illegal operation.

a. In the case of an overflow, the result is a positive or negative infinity, depending on the sign of the result.
b. In the case of an underflow, the result is a denormalized number.
c. In the case of an illegal operation (infinity values of the opposite sign have been added, an infinity has been subtracted from another infinity of the same sign, zero has been multiplied by infinity, zero is divided by zero, or infinity is divided by infinity) the result is a not-a-number.
d. If an overflow results from conversion of a floating-point number to an integer, the result is not guaranteed.

Note: Operations are performed on constant expressions during compilation. If an overflow, underflow, or illegal operation occurs, a warning level error message is output.

iii. Notes on operations on special values

The following are notes on operations on special values (zero, infinity, and not-a-number).

a. The sum of a positive zero and a negative zero is a positive zero.
b. The difference between two zeros of the same sign is a positive zero.
c. The result of operations that include not-a-number in one or both operands is always a not-a-number.
d. In comparative operations, positive zeros and negative zeros are processed as equal.
e. The result of comparative operations or equivalence operations where either one or both operands are not-a-number is true for "!=" and false in all other cases.

(b) Conversion between Decimal and Internal Representation

This section describes the specifications for conversions between floating-point numbers in a source program and internal representation, and conversion by library functions between the decimal representation of floating-point numbers in ASCII strings and their internal representation.

i. When converting from decimal to internal representation, the decimal value is first converted to its normalized form. The normalized form of a decimal value is $\pm M \times 10^{\pm N}$, where M and N are in the following range:

a. Normalized form of **float** types

$0 \leq M \leq 10^9-1$

$0 \leq N \leq 99$

b. Normalized form of **double** and **long double** types

$0 \leq M \leq 10^{17}-1$

$0 \leq N \leq 999$

If a decimal value cannot be converted to its normalized form, an overflow or underflow occurs. If the decimal representation contains more valid numerals than the normalized form, the trailing digits are truncated. In this case, a warning level error message is output during compilation and the corresponding error number is set in **errno** when the program is executed. For conversion to its normalized form, the original decimal representation must, in the form of ASCII strings, be within 511 characters. If not, an error occurs during compilation and the corresponding error number is set in **errno** when the program is executed.

When converting from internal representation to decimal, the value is first converted to the normalized decimal form, then converted to ASCII strings according to the specified format.

ii. Conversion between normalized form of decimals and internal representation

When converting from the normalized form of decimals to internal representation, and vice versa, errors cannot be avoided when the exponent is large or small. The following describes the range within which conversion is accurate, and the error limits when the values are outside that range.

a. Range for accurate conversion

The rounding shown in (a) i, "Rounding of results" is correctly applied for floating-point numbers within the ranges shown below. No overflow or underflow will occur within these ranges.

- **float** types: $0 \leq M \leq 10^9-1$, $0 \leq N \leq 13$
- **double** and **long double** types: $0 \leq M \leq 10^{17}-1$, $0 \leq N \leq 27$

b. Error limits

The difference between the error that occurs when converting values that do not fall in the ranges shown in a. above and the error that occurs when rounding is correctly performed does not exceed 0.47 times the smallest digit of the valid numerals. If the value exceeds the ranges shown in a. above, an overflow or underflow may occur during conversion. In this case, a warning level error message is output during compilation, and the corresponding error number is set in **errno** when the program is executed.

9.1.4 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 9.19 shows each operator precedence and associativity.

Table 9.19 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
12	&&	Left	Logical AND operation
13	||	Left	Logical inclusive OR expression
14	?:	Left	Conditional expression
15	= += -= *= /= %= <<= >>= &= |= ^=	Right	Assignment expression
16	,	Left	Comma expression

9.1.5 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

9.2 Extended Specifications

The compiler supports the following extended specifications:

- **#pragma** extension specifiers and keywords
- Intrinsic functions
- Section address operators

9.2.1 #pragma Extension Specifiers and Keywords

Table 9.20 lists the **#pragma** extension specifiers and keywords.

The **#pragma** extension specifiers related to optimization may not be applied depending on the condition. Confirm through the output code whether the relevant optimization has been applied.

Table 9.20 #pragma Extension Specifiers and Keywords

No.	Target	#pragma Extension Specifier*[1]	Function
1	Memory allocation	#pragma section	Switches sections.
2		#pragma stacksize	Creates a stack section.
3	Function	#pragma interrupt	Creates an interrupt function.
4		#pragma inline #pragma noinline	Performs inline expansion of a function.
5		#pragma inline_asm	Performs inline expansion of an assembly-language function.
6		#pragma entry	Creates an entry function.
7		#pragma option	Specifies options for a function.
8	Others	#pragma bit_order	Switches the order of bit assignment.
9		#pragma pack #pragma unpack #pragma packoption	Specifies the boundary alignment value for structure members and class members.
10		#pragma address	Specifies an absolute address for a variable.
11		#pragma endian	Specifies the endian for initial values.
12		__evenaccess	Guarantees access in the size of the variable type.
13		far *[2] _far *[2] near *[2] _near *[2]	Reserved keywords
14	Function	#pragma instalign4 #pragma instalign8 #pragma noinstalign	Specifies the function in which instructions at branch destinations are aligned for execution.

Notes: 1. In each **#pragma** keyword, uppercase and lowercase letters are distinguished. Therefore, if uppercase letters are used instead of lowercase letters for a keyword, warning C5161(W) will be output and the keyword will not be accepted.

2. **far**, **_far**, **near**, and **_near** are reserved for keywords. They are recognized as qualifiers but do not affect the resultant code.

#pragma section Section Switch

Format: #pragma section [<section type>] [Δ<new section name>]

<section type>: { P | C | D | B }

Description: 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 postfixed 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.

Example 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)
}
```

Example 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;
}
```

Remarks: **#pragma section** can be declared only outside the function definition.

The section name of the following items cannot be changed by this extension. The **section** option needs to be used.

(1) String literal
(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
```

#pragma stacksize Stack Section Creation

Format: #pragma stacksize {si=<constant> | su=<constant>}

Description: 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**.

Example: 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
```

Remarks: **si** and **su** can each be specified only once in a file.

<constant> must always be specified as a multiple of four.

#pragma interrupt Interrupt Function Creation

Format: #pragma interrupt [(]<function name>[(<interrupt specification>[,...])][,...][)]

Description: This extension declares an interrupt function.

A global function or a static function member can be specified for the function name.

Table 9.21 lists the interrupt specifications.

Table 9.21 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	**ACC** saving	acc	None	Saves and restores **ACC** in the interrupt function.
6	**ACC** non-saving	no_acc	None	Does not save and restore **ACC** 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 number location in the **C$VECT** section.

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 **ACC** saving is specified (**acc**), if another function is called from the specified function or the function uses an instruction that modifies the ACC, an instruction to save and restore the **ACC** is generated.

When **ACC** 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 option settings for compilation.

A global function (in C/C++ program) or a static function member (in C++ program) can be specified as an interrupt function definition.

The function must return only **void** data. No return value can be specified for the **return** statement. If attempted, an error will be output.

Example 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.
```

Example 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
```

Example 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    R1-R5          ; Saves R1 to R5.
      PUSHM    R14-R15        ; Saves R14 and R15.
      ....
      MOV.L    #_sub,R15
      JSR      R15            ; Function call
      ....
      POPM     R14-R15        ; Restores R14 and R15.
      POPM     R1-R5          ; Restores R1 to R5.
      RTE
```

Example 4: Use of interrupt specification **fint**
C source description: Compiles with the **fint_register=2** option specified

```
#pragma interrupt func(fint)
void func1(){ .... }   // Interrupt function
void func2(){ .... }   // 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.
      RTE

_func2:
      ....               ; In the functions without #pragma interrupt fint
      ....               ; specification, do not use R12 and R13.
```

Example 5: Use of interrupt specification **acc**

```
void func(void);
#pragma interrupt accsaved_ih(acc)  /* Specifies acc. */
void accsaved_ih(void)
{
     func();
}
```

Output code:

```
_accsaved_ih:                   ; function: accsaved_ih
      .STACK   _accsaved_ih=44  ; Includes ACC saved data (8 bytes)
      PUSHM    R1-R5
      PUSHM    R14-R15
      MVFACMI  R4                ; ACC saved code (1/4)
      SHLL     #10H,R4           ; ACC saved code (2/4)
      MVFACHI  R5                ; ACC saved code (3/4)
      PUSHM    R4-R5             ; ACC saved code (4/4)
      BSR      _func
      POPM     R4-R5             ; ACC restored code (1/3)
      MVTACLO  R4                ; ACC restored code (2/3)
      MVTACHI  R5                ; ACC restored code (3/3)
      POPM     R14-R15
      POPM     R1-R5
      RTE
```

Remarks: Do not specify a **static** function because it may be deleted by optimization.

Due to the specifications of the RX instruction set, 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 and 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 in section 5.2.2, Output Options.

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.

#pragma inline, #pragma noinline Inline Expansion of Function

Format: #pragma inline [(]<function name>[,...][)]
#pragma noinline [(]<function name>[,...][)]

Description: **#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.

A function specified by **#pragma inline** or a function with specifier **inline** (C++ and C (C99)) are 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;
}
```

Remarks: Inline expansion will not be applied in the following functions even when **#pragma inline** is specified.

- The function has variable parameters.
- A parameter address is referred to in the function.
- 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 even for a function specified by **#pragma inline**.

When **#pragma inline** is specified for a **static** function, the function definition is deleted after inline expansion.

No external definition will be created for functions for which **inline** (C++ and C (C99)) is specified.

#pragma inline_asm Inline Expansion of Assembly-Language Function

Format: #pragma inline_asm[(]<function name>[,...][)]

Description: 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       ; Inline expansion
        MOV.L   R1,[R6]
        POP     R6
        RTS
```

Remarks: Specify **#pragma inline_asm** before defining a function.

An external definition is generated even for a function specified by **#pragma inline_asm**.

When the registers whose values are saved and restored at the entry and exit of a function (see table 8.2) are used in an assembly-language inline function, these registers must be saved and restored at the start and end of the function.

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**).

#pragma entry Entry Function Creation

Format: #pragma entry[(]<function name>[)]

Description: 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
```

Remarks: Be sure to specify **#pragma entry** before declaring a function.

Do not specify more than one entry function in a load module.

#pragma option Option Specification for Each Function

Format: #pragma option [<option string>]

Description: This extension applies the options specified in <option string>.

The specified options are applied until the end of the file or a **#pragma option** without <option string> is specified.

When **#pragma option <option string>** is specified, optimization specified in <option string> is applied. Table 9.22 shows the optimize options that can be specified. For each option, refer to section 2, C/C++ Compiler Options.

Table 9.22 Optimize Options Specifiable in #pragma Option

No.	Option Specification	Option Cancellation
1	const_div	noconst_div
2	optimize = {0 | 1 | 2}	None
3	speedsize	sizespeed
4	loop=n (n is an integer from 2 to 32)	loop=1
5	case={ ifthen | table | auto }	None
6	schedule	noschedule
7	scope	noscope

Example: C source description: No compiler option is specified (default state)

```
#pragma option speed
void func1(){ ... }   /* Default + -speed are applied                */
#pragma option optimize=0
void func2(){ ... }   /* Default + -speed + -optimize=0 are applied */
#pragma option
void func3(){ ... }   /* Default optimization is applied             */
```

#pragma bit_order Bit Field Order Specification

Format: #pragma bit_order [{left | right}]

Description: 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**.

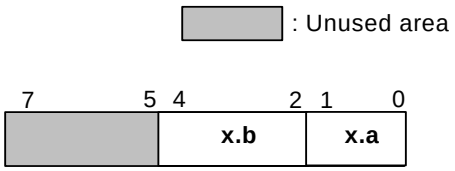
If **left** or **right** is omitted, the order is determined by the option specification.

Example:

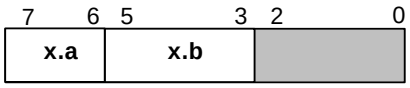
C Source

```
#pragma bit_order right
struct tbl_r {
    unsigned char a:2;
    unsigned char b:3;
} x;
```

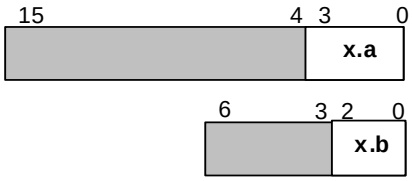
Bit Assignment



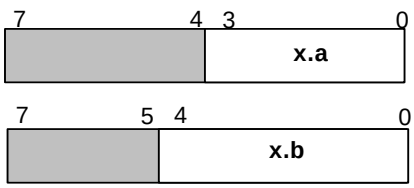

```
#pragma bit_order left
struct tbl_l {
    unsigned char a:2;
    unsigned char b:3;
} y;
```




```
// 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;
```



#pragma pack, #pragma unpack, #pragma packoption — Alignment Value Specification for Structure Members and Class Members

Format:
#pragma pack
#pragma unpack
#pragma packoption

Description: **#pragma pack** specifies the boundary alignment value for structure members and class members after the **#pragma pack** written in the source program.

When **#pragma pack** is not specified or after **#pragma packoption** is specified, the boundary alignment value for the structure members and class members is determined by the **pack** option. Table 9.23 shows **#pragma pack** specifications and the corresponding alignment values.

Table 9.23 #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         */
```

Remarks: The structure or class member for **#pragma pack** is specified cannot be accessed using a pointer (including an access within a member function using a pointer).

Example:

```
#pragma pack
struct st {
   char x;
   int y;
} ST;
int *p=&ST.y; /* The ST.y address may be an odd value.        */
void func(void) {
   ST.y=1;    /* Can be accessed correctly.                   */
   *p=1;      /* Cannot be accessed correctly in some cases.  */
}
```

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.

#pragma address Absolute Address Specification

Format: #pragma address [(]<variable name>=<absolute address>[,...][)]

Description: 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       #32512,R14      ; 0x7f00
     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.

If **#pragma section** is specified together with **#pragma address** for a single variable, an error will be output.

#pragma endian — Endian Specification for Initial Values

Format: #pragma endian [{big | little}]

Description: 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:

```
        .SECTION        D_B,ROMDATA,ALIGN=4
        .ENDIAN         BIG
        .glb            A
_A:                                 ; static: A
        .LWORD          00000064H
        .SECTION        D,ROMDATA,ALIGN=4
        .glb        _B
_B:                                 ; static: B
        .LWORD          000000C8H
```

Remarks: 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. If a code that writes to such an area is included, a warning message is displayed.

The endian of the following items cannot be changed by this extension. The **endian** option needs to be used.

(1) String literal
(2) Branch table of **switch** statement
(3) Object declared as an external reference
(object declared through **extern** without initialization expression)

__evenaccess Guarantee of Access in Specified Size

Format: __evenaccess <type specifier> <variable name>
<type specifier> __evenaccess <variable name>

Description: This extension guarantees access in the size of the target variable.

Access size is guaranteed for 4-byte or smaller scalar integer types (**signed char**, **unsigned char**, **signed short**, **unsigned short**, **signed int**, **unsigned int**, **signed long**, and **unsigned long**).

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
```

Remarks: 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.

#pragma instalign4
#pragma instalign8
#pragma noinstalign — Specification of Function in which Instructions at Branch Destinations are Aligned for Execution

Format:
#pragma instalign4 [(]<function name>[(<branch destination type>)][,...][)]
#pragma instalign8 [(]<function name>[(<branch destination type>)][,...][)]
#pragma noinstalign [(]<function name>[,...][)]

Description: 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.

9.2.2 Intrinsic Functions

The compiler provides the following intrinsic functions.

- Maximum and minimum value selection
- Byte switching in data
- Data exchange
- Multiply-and-accumulate operation
- Rotation
- Special instructions (**BRK**, **WAIT**, **INT**, and **NOP**)
- Special instructions for the RX family (such as **BRK** and **WAIT**)
- Control register setting and reference

Intrinsic functions can be written in the same call format as regular functions.

Table 9.24 lists intrinsic functions.

Table 9.24 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
2		signed long min(signed long data1, signed long data2)	Selects the minimum value.	O
3	Byte switch	unsigned long revl(unsigned long data)	Reverses the byte order in longword data.	O
4		unsigned long revw(unsigned long data)	Reverses the byte order in longword data in word units.	O
5	Data exchange	void xchg(signed long *data1, signed long *data2)	Exchanges data.	O

No.	Item	Specifications	Function	Restriction in User Mode*
6	Multiply-and-accumulate operation	long long rmpab(long long init, unsigned long count, signed char *addr1, signed char *add2)	Multiply-and-accumulate operation (byte).	O
7		long long rmpaw(long long init, unsigned long count, short *addr1, short *add2)	Multiply-and-accumulate operation (word).	O
8		long long rmpal(long long init, unsigned long count, long *addr1, long *add2)	Multiply-and-accumulate operation (longword).	O
9	Rotation	unsigned long rolc(unsigned long data)	Rotates data including the carry to left by one bit.	O
10		unsigned long rorc(unsigned long data)	Rotates data including the carry to right by one bit.	O
11		unsigned long rotl(unsigned long data, unsigned long num)	Rotates data to left.	O
12		unsigned long rotr(unsigned long data, unsigned long num)	Rotates data to right.	O
13	Special instructions	void brk(void)	**BRK** instruction exception.	O
14		void int_exception(signed long num)	**INT** instruction exception.	O
15		void wait(void)	Stops program execution.	×
16		void nop(void)	Expanded to a **NOP** instruction.	O
17	Processor interrupt priority level (IPL)	void set_ipl(signed long level)	Sets the interrupt priority level.	×
18		unsigned char get_ipl(void)	Refers to the interrupt priority level.	O
19	Processor status word (PSW)	void set_psw(unsigned long data)	Sets data to **PSW**.	Δ
20		unsigned long get_psw(void)	Refers to **PSW** value.	O
21	Floating-point status word (FPSW)	void set_fpsw(unsigned long data)	Sets data to **FPSW**.	O
22		unsigned long get_fpsw(void)	Refers to **FPSW** value.	O

No.	Item	Specifications	Function	Restriction in User Mode*
23	User stack pointer (USP)	void set_usp(void * data)	Sets data to **USP**.	O
24		void * get_usp(void)	Refers to **USP** value.	O
25	Interrupt stack pointer (ISP)	void set_isp(void * data)	Sets data to **ISP**.	Δ
26		void * get_isp(void)	Refers to **ISP** value.	O
27	Interrupt table register (INTB)	void set_intb(void * data)	Sets data to **INTB**.	Δ
28		void * get_intb(void)	Refers to **INTB** value.	O
29	Backup PSW (BPSW)	void set_bpsw(unsigned long data)	Sets data to **BPSW**.	Δ
30		unsigned long get_bpsw(void)	Refers to **BPSW** value.	O
31	Backup PC (BPC)	void set_bpc(void * data)	Sets data to **BPC**.	Δ
32		void * get_bpc(void)	Refers to **BPC** value.	O
33	Fast interrupt vector register (FINTV)	void set_fintv(void * data)	Sets data to **FINTV**.	Δ
34		void * get_fintv(void)	Refers to **FINTV** value.	O
35	Significant 64-bit multiplication	signed long long emul(signed long data1, signed long data2)	Signed multiplication of significant 64 bits.	O
36		unsigned long long emulu(unsigned long data1, unsigned long data2)	Unsigned multiplication of significant 64 bits.	O
37	Processor mode (**PM**)	void chg_pmusr(void)	Switches to user mode.	Δ
38	Accumulator (**ACC**)	void set_acc(signed long long data)	Sets the **ACC**.	O
39		signed long long get_acc(void)	Refers to the **ACC**.	O
40	Control of the interrupt enable bits	void setpsw_i(void)	Sets the interrupt enable bit to 1.	Δ
41		void clrpsw_i(void)	Clears the interrupt enable bit to 0.	Δ
42	Multiply-and-accumulate operation	long macl(short* data1, short* data2, unsigned long count)	Multiply-and-accumulate operation of 2-byte data.	O
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

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) Selection of Maximum Value

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.
}
```

signed long min(signed long data1, signed long data2) Selection of Minimum Value

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.
}
```

unsigned long revl(unsigned long data) Byte Order Reversal in Longword Data

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
}
```

unsigned long revw(unsigned long data)

Byte Order Reversal in Longword Data in Word Units

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
}
```

void xchg(signed long *data1, signed long *data2) Data Exchange

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.
}
```

long long rmpab(long long init, unsigned long count, signed char *addr1, signed char *add2) Multiply-and-Accumulate Operation (Byte)

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.
}
```

Remarks: The **RMPA** instruction obtains a result in a maximum of 80 bits, but this intrinsic function handles only 64 bits.

long long rmpaw(long long init, unsigned long count, short *addr1, short *add2) — Multiply-and-Accumulate Operation (Word)

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.
}
```

Remarks: The **RMPA** instruction obtains a result in a maximum of 80 bits, but this intrinsic function handles only 64 bits.

long long rmpal(long long init, unsigned long count, long *addr1, long *add2) — Multiply-and-Accumulate Operation (Longword)

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.
}
```

unsigned long rolc(unsigned long data) One-Bit Left Rotation Including Carry

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.
}
```

unsigned long rorc(unsigned long data) One-Bit Right Rotation Including Carry

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.
}
```

unsigned long rotl(unsigned long data, unsigned long num) Left Rotation

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.
}
```

unsigned long rotr(unsigned long data, unsigned long num) Right Rotation

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.
}
```

void brk(void) BRK Instruction Exception

Description: This function is expanded into a **BRK** instruction.

Header: <machine.h>

Example:

```
#include <machine.h>
void main(void)
{
     brk();                    // BRK instruction
}
```

void int_exception(signed long num) INT Instruction Exception

Description: This function is expanded into an **INT num** instruction.

Header: <machine.h>

Parameters: num **INT** instruction number

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**.

void wait(void) Program Execution Stop

Description: This function is expanded into a **WAIT** instruction.

Header: <machine.h>

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.

void nop(void) Expansion to NOP Instruction

Description: This function is expanded into a **NOP** instruction.

Header: <machine.h>

Example:

```
#include <machine.h>
void main(void)
{
     nop();                     // NOP instruction
}
```

void set_ipl(signed long level) Interrupt Priority Level Setting

Description: Changes the interrupt mask level.

Header: <machine.h>

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.

unsigned char get_ipl(void) Interrupt Priority Level Reference

Description: Refers to the interrupt mask level.

Header: <machine.h>

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.

void set_psw(unsigned long data) PSW Setting

Description: Sets a value to **PSW**.

Header: <machine.h>

Parameters: data Value to be set

Example:

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
     set_psw(data);      // Sets PSW to a 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.

unsigned long get_psw(void) PSW Reference

Description: Refers to the **PSW** value.

Header: <machine.h>

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.

void set_fpsw(unsigned long data) FPSW Setting

Description: Sets a value to **FPSW**.

Header: <machine.h>

Parameters: data Value to be set

Example:

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
     set_fpsw(data);     // Sets FPSW to a value specified by data.
}
```

unsigned long get_fpsw(void) FPSW Reference

Description: Refers to the **FPSW** value.

Header: <machine.h>

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.

void set_usp(void * data) USP Setting

Description: Sets a value to **USP**.

Header: <machine.h>

Parameters: data Value to be set

Example:

```
#include <machine.h>
extern void * data;
void main(void)
{
     set_usp(data);       // Sets USP to a value specified by data.
}
```

void * get_usp(void) USP Reference

Description: Refers to the **USP** value.

Header: <machine.h>

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.
}
```

void set_isp(void * data) ISP Setting

Description: Sets a value to **ISP**.

Header: <machine.h>

Parameters: data Value to be set

Example:

```
#include <machine.h>
extern void * data;
void main(void)
{
     set_isp(data);      // Sets ISP to a 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.

void * get_isp(void) ISP Reference

Description: Refers to the **ISP** value.

Header: <machine.h>

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.
}
```

void set_intb (void * data) INTB Setting

Description: Sets a value to **INTB**.

Header: <machine.h>

Parameters: data Value to be set

Example:

```
#include <machine.h>
extern void * data;
void main(void)
{
     set_intb (data);    // Sets INTB to a 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.

void * get_intb(void) INTB Reference

Description: Refers to the **INTB** value.

Header: <machine.h>

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.
}
```

void set_bpsw(unsigned long data) BPSW Setting

Description: Sets a value to **BPSW**.

Header: <machine.h>

Parameters: data Value to be set

Example:

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
     set_bpsw (data);    // Sets BPSW to a 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.

unsigned long get_bpsw(void) BPSW Reference

Description: Refers to the **BPSW** value.

Header: <machine.h>

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.
}
```

void set_bpc(void * data) BPC Setting

Description: Sets a value to **BPC**.

Header: <machine.h>

Parameters: data Value to be set

Example:

```
#include <machine.h>
extern void * data;
void main(void)
{
     set_bpc(data);      // Sets BPC to a 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.

void * get_bpc(void) BPC Reference

Description: Refers to the **BPC** value.

Header: <machine.h>

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.
}
```

void set_fintv(void * data) FINTV Setting

Description: Sets a value to **FINTV**.

Header: <machine.h>

Parameters: data Value to be set

Example:

```
#include <machine.h>
extern void * data;
void main(void)
{
     set_fintv(data);    // Sets FINTV to a 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.

void * get_fintv(void) FINTV Reference

Description: Refers to the **FINTV** value.

Header: <machine.h>

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.
}
```

signed long long emul(signed long data1, signed long data2) 64-Bit Signed Multiplication

Description: Performs signed multiplication of significant 64 bits.

Header: <machine.h>

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.
}
```

unsigned long long emulu(unsigned long data1, unsigned long data2) 64-Bit Unsigned Multiplication

Description: Performs unsigned multiplication of significant 64 bits.

Header: <machine.h>

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.
}
```

void chg_pmusr(void) Switching to User Mode

Description: Switches the RX processor mode to user mode.

Header: <machine.h>

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.

void set_acc(signed long long data) ACC Setting

Description: Sets a value to **ACC**.

Header: <machine.h>

Parameters: data Value to be set to **ACC**

Example:

```
#include <machine.h>
void main(void)
{
     signed long long data = 0x123456789ab0000LL;
     set_acc(data);      // Sets ACC to a value specified by data.
}
```

signed long long get_acc(void) ACC Reference

Description: Refers to the **ACC** value.

Header: <machine.h>

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 func(signed long a, signed long b)
{
     signed long long bak_acc = get_acc();
                          // Obtains the ACC value and saves it
                          // in bak_acc.
     a *= b;              // Multiplication (ACC is damaged).
     set_acc(bak_acc);    // Restores ACC with a value saved by
                          // bak_acc.
     return a;
}
```

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.

void setpsw_i(void) Interrupt Enable Bit Setting to 1

Description: Sets the interrupt enable bit (I bit) in PSW to 1.

Header: <machine.h>

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.

void clrpsw_i(void) Interrupt Enable Bit Clearing to 0

Description: Clears the interrupt enable bit (I bit) in PSW to 0.

Header: <machine.h>

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.

long macl(short * data1, short * data2, unsigned long count)

Multiply-and-Accumulate Operation (2 Bytes)

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 **MVFACHI** 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**.

short macw1(short* data1, short* data2, unsigned long count) **Multiply-and-Accumulate**
short macw2(short* data1, short* data2, unsigned long count) **Operation (Fixed-Point)**

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**.

9.2.3 Section Address Operators

Table 9.25 lists the section address operators.

Table 9.25 Section Address Operators

No.	Section Address Operator	Description
1	__sectop("<section name>")	Refers to the start address of the specified <section name>.
2	__secend("<section name>")	Refers to the end address + 1 of the specified <section name>.
3	__secsize("<section name>")	Generates the size of the specified <section name>.

__sectop, __secend, __secsize Section Address Operators

Format: __sectop("<section name>")
__secend("<section name>")
__secsize("<section name>")

Description: **__sectop** refers to the start address of the specified <section name>.
__secend refers to the end address + 1 of the specified <section name>.
__secsize generates the size of the specified <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 section 2.5, Microcontroller Options, and section 8.4, Usage of PIC/PID Function.

9.3 C/C++ Libraries

9.3.1 Standard C Libraries

(1) Overview of Libraries

This section describes the specifications of the C library functions, which can be used generally in C/C++ programs. This section gives an overview of the library configuration, and describes the layout and the terms used in this library function description.

Note: The description in this section assumes that the **dbl_size=8** compiler option is specified for the **double** and **long double** types.

(a) Library Types

A library implements standard processing such as input/output and string handling in the form of C/C++ language functions. Libraries can be used by including standard include files for each unit of processing.

Standard include files contain declarations for the corresponding libraries and definitions of the macro names necessary to use them.

Table 9.26 shows the various library types and the corresponding standard include files.

Table 9.26 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 9.27, are provided to improve programming efficiency.

Table 9.27 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.

(b) Organization of Library Part

The organization of the library part of this manual is described below.

Library functions are categorized according to the corresponding standard include file, and descriptions are given for each standard include file. For each category, there is first a description relating to the macro names and function declarations defined in the standard include file (figure 9.2), followed by a description of each function (figure 9.3).

Figure 9.2 shows the standard include file description layout, and figure 9.3, the function description layout.

Section number <standard include file name>

- Gives a functional overview of this standard include file.
- Describes the names defined or declared in this standard include file with classifying them by name type such as [Type], [Constant], [Variable], and [Function]. For macro names, (macro) is always attached beside the name type or name description.
- Adds description if implementation-defined specifications are included or notes common to the functions declared in this standard include file are given.

Figure 9.2 Layout of Standard Include File Description

Function type and name (return value and parameters)	Functional overview
Description:	Describes the library function.
Header file:	Shows the name of standard include file that contains this function declaration.
Return value:	Normal: Shows the return value when the library function ends normally. Abnormal: Shows the return value when the library function ends abnormally.
Parameters:	Indicates the meanings of the parameters.
Example:	Describes the calling procedure.
Error conditions:	Conditions for the occurrence of errors that cannot be determined from the return value in library function processing. If such an error occurs, the value defined in each compiler for the error type is set in **errno***.
Remarks:	Provides summplementary information or notes on usage.
Implementation define:	Describes the processing method in this compiler

Figure 9.3 Layout of Function Description

Note: **errno** is a variable that stores the error type if an error occurs during execution of a library function. See section 9.3.1 (2), <stddef.h>, for details.

(c) Terms Used in Library Function Descriptions

(i) 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.

(ii) **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.

(iii) 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.

(iv) **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.

(v) **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.

(vi) 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.

(vii) 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.

(viii) 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.

(ix) 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.

(x) 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.

Example: 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.

(xi) 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 9.28.

Table 9.28 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

(xii) Implementation definition

Definitions differ for each compiler.

(xiii) 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.

(xiv) 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.

(d) 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.

(2) <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 (macro)	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 (macro)	wchar_t	Type that indicates an extended character.

Implementation-Defined Specifications

Item	Compiler Specifications
Value of macro **NULL**	Value 0 (pointer to **void**)
Type equivalent to macro **ptrdiff_t**	**long** type
Type equivalent to **wchar_t**	**short** type

(3) <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.

void assert (long expression) Diagnostics

Description: Adds diagnostics into programs.

Header file: <assert.h>

Parameters: expression Expression to be evaluated.

Example:
```
#include <assert.h>
int expression;
    assert (expression);
```

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:Δexpression ΔFILE Δ<file name>,
LINE Δ<line number>
```

(2) When **-lang=c99** is specified (C (C99) language):

```
ASSERTION FAILED:Δexpression ΔFILE Δ<file name>,
LINE Δ<line number> ΔFUNCNAME Δ<function name>
```

(4) <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	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 9.29.

Table 9.29 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
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. Table 9.30 shows the character set that results in a true return value.

Table 9.30 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'

long isalnum (long c) — Test for Letter or Decimal Digit

Description: Tests for a letter or a decimal digit.

Header file: <ctype.h>

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

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=isalnum(c);
```

long isalpha(long c) — Test for Letter

Description: Tests for a letter.

Header file: <ctype.h>

Return values: If character **c** is a letter: Nonzero
If character **c** is not a letter: 0

Parameters: c Character to be tested

Example:

```
#include <ctype.h>
int c, ret;
    ret=isalpha(c);
```

long iscntrl (long c) Test for Control Character

Description: Tests for a control character.

Header file: <ctype.h>

Return values: If character **c** is a control character: Nonzero
If character **c** is not a control character: 0

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=iscntrl (c);
```

long isdigit (long c) Test for Decimal Digit

Description: Tests for a decimal digit.

Header file: <ctype.h>

Return values: If character **c** is a decimal digit: Nonzero
If character **c** is not a decimal digit: 0

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=isdigit(c);
```

long isgraph (long c) Test for Printing Character Except Space

Description: Tests for any printing character except space (‘ ’).

Header file: <ctype.h>

Return values: If character **c** is a printing character except space: Nonzero
If character **c** is not a printing character except space: 0

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=isgraph(c);
```

long islower (long c) Test for Lowercase Letter

Description: Tests for a lowercase letter.

Header file: <ctype.h>

Return values: If character **c** is a lowercase letter: Nonzero
If character **c** is not a lowercase letter: 0

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=islower(c);
```

long isprint (long c) Test for Printing Character

Description: Tests for a printing character including space (‘ ’).

Header file: <ctype.h>

Return values: If character **c** is a printing character including space: Nonzero
If character **c** is not a printing character including space: 0

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=isprint(c);
```

long ispunct (long c) Test for Special Character

Description: Tests for a special character.

Header file: <ctype.h>

Return values: If character **c** is a special character: Nonzero
If character **c** is not a special character: 0

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=ispunct(c);
```

long isspace (long c) Test for White-Space Character

Description: Tests for a white-space character.

Header file: <ctype.h>

Return values: If character **c** is a white-space character: Nonzero
If character **c** is not a white-space character: 0

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=isspace(c);
```

long isupper (long c) Test for Uppercase Letter

Description: Tests for an uppercase letter.

Header file: <ctype.h>

Return values: If character **c** is an uppercase letter: Nonzero
If character **c** is not an uppercase letter: 0

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=isupper(c);
```

long isxdigit (long c) Test for Hexadecimal Digit

Description: Tests for a hexadecimal digit.

Header file: <ctype.h>

Return values: If character **c** is a hexadecimal digit: Nonzero
If character **c** is not a hexadecimal digit: 0

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=isxdigit(c);
```

long tolower (long c) Conversion to Lowercase Letter

Description: Converts an uppercase letter to the corresponding lowercase letter.

Header file: <ctype.h>

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**

Parameters: c Character to be converted

Example:
```
#include <ctype.h>
int c, ret;
    ret=tolower(c);
```

long toupper (long c) Conversion to Uppercase Letter

Description: Converts a lowercase letter to the corresponding uppercase letter.

Header file: <ctype.h>

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**

Parameters: c Character to be converted

Example:
```
#include <ctype.h>
int c, ret;
    ret=toupper(c);
```

long isblank (long c) Test for Blank

Description: Tests for a space character or a tab character.

Header file: <ctype.h>

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

Parameters: c Character to be tested

Example:
```
#include <ctype.h>
int c, ret;
    ret=isblank(c);
```

(5) <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 result of an add operation is rounded off. The meaning of this macro definition is as follows: • When result of add operation is rounded off: Positive value • When result of add operation is rounded down: 0 • When nothing is specified: –1 The rounding-off and rounding-down methods are implementation-defined.
	FLT_GUARD	1	Indicates whether or not a guard bit is used in multiply operations. The meaning of this macro definition is as follows: • When guard bit is used: 1 • When guard bit is not used: 0
	FLT_NORMALIZE	1	Indicates whether or not floating-point values are normalized. The meaning of this macro definition is as follows: • When normalized: 1 • When not normalized: 0
	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.

Type	Definition Name	Definition Value	Description
Constant (macro)	LDBL_MAX	1.7976931348623158e +308	Indicates the maximum value that can be represented as a **long double** type floating-point value.
	FLT_MAX_EXP	127	Indicates the power-of-radix maximum value that can be represented as a **float** type floating-point value.
	DBL_MAX_EXP	1023	Indicates the power-of-radix maximum value that can be represented as a **double** type floating-point value.
	LDBL_MAX_EXP	1023	Indicates the power-of-radix maximum value that can be represented as a **long double** type floating-point value.
	FLT_MAX_10_ EXP	38	Indicates the power-of-10 maximum value that can be represented as a **float** type floating-point value.
	DBL_MAX_10_ EXP	308	Indicates the power-of-10 maximum value that can be represented as a **double** type floating-point value.
	LDBL_MAX_10_ EXP	308	Indicates the power-of-10 maximum value that can be represented as a **long double** type floating-point value.
	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.
	FLT_MIN_EXP	−149	Indicates the power-of-radix minimum value of a floating-point value that can be represented as a **float** type positive value.
	DBL_MIN_EXP	−1074	Indicates the power-of-radix minimum value of a floating-point value that can be represented as a **double** type positive value.

Type	Definition Name	Definition Value	Description
Constant (macro)	LDBL_MIN_EXP	–1074	Indicates the power-of-radix minimum value of a floating-point value that can be represented as a **long double** type positive value.
	FLT_MIN_10_EXP	–44	Indicates the power-of-10 minimum value of a floating-point value that can be represented as a **float** type positive value.
	DBL_MIN_10_ EXP	–323	Indicates the power-of-10 minimum value of a floating-point value that can be represented as a **double** type positive value.
	LDBL_MIN_10_ EXP	–323	Indicates the power-of-10 minimum value of a floating-point value that can be represented as a **long double** type positive value.
	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.

Type	Definition Name	Definition Value	Description
Constant (macro)	FLT_EXP_DIG	8	Indicates the maximum number of exponent digits when a **float** type floating-point value is represented in the radix.
	DBL_EXP_DIG	11	Indicates the maximum number of exponent digits when a **double** type floating-point value is represented in the radix.
	LDBL_EXP_DIG	11	Indicates the maximum number of exponent digits when a **long double** type floating-point value is represented in the radix.
	FLT_POS_EPS	5.9604648328104311e−8F	Indicates the minimum floating-point value **x** for which 1.0 + x ≠ 1.0 in **float** type.
	DBL_POS_EPS	1.1102230246251567e−16	Indicates the minimum floating-point value **x** for which 1.0 + x ≠ 1.0 in **double** type.
	LDBL_POS_EPS	1.1102230246251567e−16	Indicates the minimum floating-point value **x** for which 1.0 + x ≠ 1.0 in **long double** type.
	FLT_NEG_EPS	2.9802324164052156e−8F	Indicates the minimum floating-point value **x** for which 1.0 – x ≠ 1.0 in **float** type.
	DBL_NEG_EPS	5.5511151231257834e−17	Indicates the minimum floating-point value **x** for which 1.0 – x ≠ 1.0 in **double** type
	LDBL_NEG_EPS	5.5511151231257834e−17	Indicates the minimum floating-point value **x** for which 1.0 – x ≠ 1.0 in **long double** type.
	FLT_POS_EPS_EXP	−23	Indicates the minimum integer **n** for which 1.0 + $(radix)^n$ ≠ 1.0 in **float** type.
	DBL_POS_EPS_EXP	−52	Indicates the minimum integer **n** for which 1.0 +$(radix)^n$ ≠ 1.0 in **double** type.
	LDBL_POS_EPS_EXP	−52	Indicates the minimum integer **n** for which 1.0 + $(radix)^n$ ≠ 1.0 in **long double** type.
	FLT_NEG_EPS_EXP	−24	Indicates the minimum integer **n** for which 1.0 – $(radix)^n$ ≠ 1.0 in **float** type.

Type	Definition Name	Definition Value	Description
Constant (macro)	DBL_NEG_EPS_EXP	–53	Indicates the minimum integer **n** for which $1.0 - (radix)^n \neq 1.0$ in **double** type.
	LDBL_NEG_EPS_EXP	–53	Indicates the minimum integer **n** for which $1.0 - (radix)^n \neq 1.0$ in **long double** type.
	DECIMAL_DIG	10	Indicates the maximum number of digits of a floating-point value represented in decimal precision.
	FLT_EPSILON	1E–5	Indicates the difference between 1 and the minimum value greater than 1 that can be represented in **float** type.
	DBL_EPSILON	1E–9	Indicates the difference between 1 and the minimum value greater than 1 that can be represented in **double** type.
	LDBL_EPSILON	1E–9	Indicates the difference between 1 and the minimum value greater than 1 that can be represented in **long double** type.

(6) <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 minimum 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 32767*[2]	Indicates the maximum value that can be represented by an **int** type variable.
	INT_MIN	–2147483647–1 –32768*[2]	Indicates the minimum value that can be represented by an **int** type variable.
	UINT_MAX	4294967295U 65535U*[2]	Indicates the maximum value that can be represented by an **unsigned int** type variable.

Type	Definition Name	Definition Value	Description
Constant (macro)	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.
2. Indicates the value that can be represented by a variable when the **int_to_short** option is specified.

(7) <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 11.3, 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	

(8) <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 HUGE_VALL	Indicates the value for the function return value if the result of a function overflows.
	INFINITY	Expanded to a **float**-type constant expression that represents positive or unsigned infinity.
	NAN	Defined when **float**-type **qNaN** is supported.
	FP_INFINITE FP_NAN FP_NORMAL FP_SUBNORMAL FP_ZERO	These indicate exclusive types of floating-point values.
	FP_FAST_FMA FP_FAST_FMA FFP_FAST_FMAFL	Defined when the **Fma** function is executed at the same or higher speed than a multiplication and an addition with **double**-type operands.
	FP_ILOGB0 FP_ILOGBNAN	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 MATH_ERREXCEPT	These are expanded to integer constants 1 and 2, respectively.
	math_errhandling	Expanded to an **int**-type expression whose value is a bitwise logical OR of **MATH_ERRNO** and **MATH_ERREXCEPT**.
Type	float_t double_t	These are floating-point types having the same width as **float** and **double**, respectively.

Type	Definition Name	Description
Function (macro)	fpclassify	Classifies argument values into not-a-number, infinity, normalized number, denormalized number, and 0.
	isfinite	Determines whether the argument is a finite value.
	isinf	Determines whether the argument is infinity.
	isnan	Determines whether the argument is a not-a-number.
	isnormal	Determines whether the argument is a normalized number.
	signbit	Determines whether the sign of the argument is negative.
	isgreater	Determines whether the first argument is greater than the second argument.
	isgreaterequal	Determines whether the first argument is equal to or greater than the second argument.
	isless	Determines whether the first argument is smaller than the second argument.
	Islessequal	Determines whether the first argument is equal to or smaller than the second argument.
	Islessgreater	Determines whether the first argument is smaller or greater than the second argument.
	Isunordered	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.

Type	Definition Name	Description
Function	sin	Calculates the sine of a floating-point radian value.
	sinf	
	sinl	
	tan	Calculates the tangent of a floating-point radian value.
	tanf	
	tanl	
	cosh	Calculates the hyperbolic cosine of a floating-point number.
	coshf	
	coshl	
	sinh	Calculates the hyperbolic sine of a floating-point number.
	sinhf	
	sinhl	
	tanh	Calculates the hyperbolic tangent of a floating-point number.
	tanhf	
	tanhl	
	exp	Calculates the exponential function of a floating-point number.
	expf	
	expl	
	frexp	Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.
	frexpf	
	frexpl	
	ldexp	Multiplies a floating-point number by a power of 2.
	ldexpf	
	ldexpl	
	log	Calculates the natural logarithm of a floating-point number.
	logf	
	logl	
	log10	Calculates the base-ten logarithm of a floating-point number.
	log10f	
	log10l	

Type	Definition Name	Description
Function	modf	Breaks a floating-point number into integral and fractional parts.
	modff	
	modfl	
	pow	Calculates a power of a floating-point number.
	powf	
	powl	
	sqrt	Calculates the positive square root of a floating-point number.
	sqrtf	
	sqrtl	
	ceil	Calculates the smallest integral value not less than or equal to the given floating-point number.
	ceilf	
	ceill	
	fabs	Calculates the absolute value of a floating-point number.
	fabsf	
	fabsl	
	floor	Calculates the largest integral value not greater than or equal to the given floating-point number.
	floorf	
	floorl	
	fmod	Calculates the remainder of a division of two floating-point numbers.
	fmodf	
	fmodl	
	acosh	Calculates the hyperbolic arc cosine of a floating-point number.
	acoshf	
	acoshl	
	asinh	Calculates the hyperbolic arc sine of a floating-point number.
	asinhf	
	asinhl	
	atanh	Calculates the hyperbolic arc tangent of a floating-point number.
	atanhf	
	atanhl	

Type	Definition Name	Description
Function	exp2	Calculates the value of 2 raised to the power **x**.
	exp2f	
	exp2l	
	expm1	Calculates the natural logarithm raised to the power **x** and subtracts 1 from the result.
	expm1f	
	expm1l	
	ilogb	Extracts the exponent of **x** as a signed **int** value.
	ilogbf	
	ilogbl	
	log1p	Calculates the natural logarithm of the argument + 1.
	log1pf	
	log1pl	
	log2	Calculates the base-2 logarithm.
	log2f	
	log2l	
	logb	Extracts the exponent of **x** as a signed integer.
	logbf	
	logbl	
	scalbn	Calculates **x** × **FLT_RADIXn**.
	scalbnf	
	scalbnl	
	scalbln	
	scalblnf	
	scalblnl	
	cbrt	Calculates the cube root of a floating-point number.
	cbrtf	
	cbrtl	
	hypot	Raises each floating-point number to the power 2 and calculates the sum of the resultant values.
	hypotf	
	hypotl	

Type	Definition Name	Description
Function	erf	Calculates the error function.
	erff	
	erfl	
	erfc	Calculates the complementary error function.
	erfcf	
	erfcl	
	lgamma	Calculates the natural logarithm of the absolute value of the gamma function.
	lgammaf	
	lgammal	
	tgamma	Calculates the gamma function.
	tgammaf	
	tgammal	
	nearbyint	Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.
	nearbyintf	
	nearbyintl	
	rint	Equivalent to **nearbyint** except that this function group may generate floating-point exception.
	rintf	
	rintl	
	lrint	Rounds a floating-point number to the nearest integer according to the rounding direction.
	lrintf	
	lrintl	
	llrint	
	llrintf	
	llrintl	
	round	Rounds a floating-point number to the nearest integer in the floating-point representation.
	roundf	
	roundl	

Type	Definition Name	Description
Function	lround lroundf lroundl llround llroundf llroundl	Rounds a floating-point number to the nearest integer.
	trunc truncf truncl	Rounds a floating-point number to the nearest integer.
	remainder remainderf remainderl	Calculates remainder **x REM y** specified in the IEEE60559 standard.
	remquo remquof remquol	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	Generates a value consisting of the given absolute value and sign.
	nan nanf nanl	**nan("n string")** is equivalent to **("NAN(n string)", (char**) NULL)**.
	nextafter nextafterf nextafterl	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	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	Calculates the positive difference.

Type	Definition Name	Description
Function	fmax	Obtains the greater of two values.
	fmaxf	
	fmaxl	
	fmin	Obtains the smaller of two values.
	fminf	
	fminl	
	fma	Calculates **(d1 * d2) + d3** as a single ternary operation.
	fmaf	
	fmal	

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.

Example:

```
          .
          .
          .
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.

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 9.1.3, 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 **fmod** function is 0	A range error occurs.

double acos (double d)
float acosf (float d)
long double acosl (long double d) Arc Cosine

Description: Calculates the arc cosine of a floating-point number.

Header file: <math.h>

Return values:
Normal: Arc cosine of **d**
Abnormal: Domain error: Returns not-a-number.

Parameters:
d Floating-point number for which arc cosine is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=acos(d);
```

Error conditions: A domain error occurs for a value of **d** not in the range [–1.0, +1.0].

Remarks: The **acos** function returns the arc cosine in the range [0, π] by the radian.

double asin (double d)
float asinf (float d)
long double asinl (long double) Arc Sine

Description: Calculates the arc sine of a floating-point number.

Header file: <math.h>

Return values: Normal: Arc sine of **d**
Abnormal: Domain error: Returns not-a-number.

Parameters: d Floating-point number for which arc sine is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=asin(d);
```

Error conditions: A domain error occurs for a value of **d** not in the range [–1.0, +1.0].

Remarks: The **asin** function returns the arc sine in the range [–π/2, +π/2] by the radian.

double atan (double d)
float atanf (float d)
long double atanl (long double d) Arc Tangent

Description: Calculates the arc tangent of a floating-point number.

Header file: <math.h>

Return values: Arc tangent of **d**

Parameters: d Floating-point number for which arc tangent is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=atan(d);
```

Remarks: The **atan** function returns the arc tangent in the range (–π/2, +π/2) by the radian.

double atan2 (double y, double x)
float atan2f (float y, float x)
long double atan2l (long double y, long double x) Arc Tangent after Division

Description: Calculates the arc tangent of the division of two floating-point numbers.

Header file: <math.h>

Return values:
Normal:	Arc tangent value when **y** is divided by **x**
Abnormal:	Domain error: Returns not-a-number.

Parameters:
x	Divisor
y	Dividend

Example:
```
#include <math.h>
double x, y, ret;
    ret=atan2(y, x);
```

Error conditions: A domain error occurs if the values of both **x** and **y** are 0.0.

Remarks: The **atan2** function returns the arc tangent in the range (–π, +π) by the radian. The meaning of the **atan2** function is illustrated in figure 9.4. 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 ±π/2, depending on whether **y** is positive or negative.

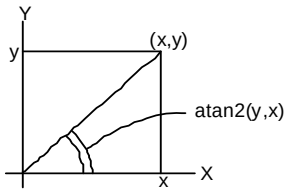


Figure 9.4 Meaning of atan2 Function

double cos (double d)
float cosf (float d)
long double cosl (long double d) Cosine

Description: Calculates the cosine of a floating-point radian value.

Header file: <math.h>

Return values: Cosine of **d**

Parameters: d Radian value for which cosine is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=cos(d);
```

double sin (double d)
float sinf (float d)
long double sinl (long double d) Sine

Description: Calculates the sine of a floating-point radian value.

Header file: <math.h>

Return values: Sine of **d**

Parameters: d Radian value for which sine is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=sin(d);
```

double tan (double d)
float tanf (float d)
long double tanl (long double d) Tangent

Description: Calculates the tangent of a floating-point radian value.

Header file: <math.h>

Return values: Tangent of **d**

Parameters: d Radian value for which tangent is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=tan(d);
```

double cosh (double d)
float coshf (float d)
long double coshl (long double d) Hyperbolic Cosine

Description: Calculates the hyperbolic cosine of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic cosine of **d**

Parameters: d Floating-point number for which hyperbolic cosine is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=cosh(d);
```

double sinh (double d)
float sinhf (float d)
long double sinhl (long double d) Hyperbolic Sine

Description: Calculates the hyperbolic sine of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic sine of **d**

Parameters:

d		Floating-point number for which hyperbolic sine is to be computed

Example:

```
#include <math.h>
double d, ret;
    ret=sinh(d);
```

double tanh (double d)
float tanhf (float d)
long double tanhl (long double d) Hyperbolic Tangent

Description: Calculates the hyperbolic tangent of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic tangent of **d**

Parameters:

d		Floating-point number for which hyperbolic tangent is to be computed

Example:

```
#include <math.h>
double d, ret;
    ret=tanh(d);
```

double exp (double d)
float expf (float d)
long double expl (long double d) Exponential Function

Description: Calculates the exponential function of a floating-point number.

Header file: <math.h>

Return values: Exponential function value of **d**

Parameters:

d	Floating-point number for which exponential function is to be computed

Example:

```
#include <math.h>
double d, ret;
    ret=exp(d);
```

double frexp (double value, double long *exp)
float frexpf (float value, long * exp) Breaking Floating-Point Number
long double frexpl (long double value, long *exp) into Mantissa and Exponent

Description: Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.

Header file: <math.h>

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

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

Example:

```
#include <math.h>
double ret, value;
long *exp;
    ret=frexp(value, exp);
```

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.

double ldexp (double e, long f)
float ldexpf (float e, long f)
long double ldexpl (long double e, long f) Converting Mantissa and Exponent into Floating-Point Number

Description: Multiplies a floating-point number by a power of 2.

Header file: <math.h>

Return values: Result of $e * 2^f$ operation

Parameters:

e	Floating-point number to be multiplied by a power of 2
f	Power-of-2 value

Example:

```
#include <math.h>
double ret, e;
int f;
    ret=ldexp(e, f);
```

double log (double d)
float logf (float d)
long double logl (long double d) Natural Logarithm

Description: Calculates the natural logarithm of a floating-point number.

Header file: <math.h>

Return values:

Normal:	Natural logarithm of **d**
Abnormal:	Domain error: Returns not-a-number.

Parameters:

d	Floating-point number for which natural logarithm is to be computed

Example:

```
#include <math.h>
double d, ret;
    ret=log(d);
```

Error conditions: A domain error occurs if **d** is negative.

A range error occurs if **d** is 0.0.

double log10 (double d)
float log10f(float d)
long double log10l(long double d) Base-Ten Logarithm

Description: Calculates the base-ten logarithm of a floating-point number.

Header file: <math.h>

Return values: Normal: Base-ten logarithm of **d**
Abnormal: Domain error: Returns not-a-number.

Parameters:

d	Floating-point number for which base-ten logarithm is to be computed

Example:

```
#include <math.h>
double d, ret;
    ret=log10(d);
```

Error conditions: A domain error occurs if **d** is negative.

A range error occurs if **d** is 0.0.

double modf (double a, double*b)
float modff (float a, float *b)
long double modfl (long double a, long double *b) Breaking Floating-Point Number into Integral and Fractional Parts

Description: Breaks a floating-point number into integral and fractional parts.

Header file: <math.h>

Return values: Fractional part of **a**

Parameters:

a	Floating-point number to be broken into integral and fractional parts
b	Pointer indicating storage area that stores integral part

Example:
```
#include <math.h>
double a, *b, ret;
    ret=modf(a, b);
```

double pow (double x, double y)
float powf (float x, float y)
long double powl (long double x, long double y) Power of Floating-Point Number

Description: Calculates a power of floating-point number.

Header file: <math.h>

Return values:

Normal:	Value of **x** raised to the power **y**
Abnormal:	Domain error: Returns not-a-number.

Parameters:

x	Value to be raised to a power
y	Power value

Example:
```
#include <math.h>
double x, y, ret;
    ret=pow(x, y);
```

Error conditions: 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.

double sqrt (double d)
float sqrtf (float d)
long double sqrtl (long double d) Square Root

Description: Calculates the positive square root of a floating-point number.

Header file: <math.h>

Return values:

Normal:	Positive square root of **d**
Abnormal:	Domain error: Returns not-a-number.

Parameters:

d	Floating-point number for which positive square root is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=sqrt(d);
```

Error conditions: A domain error occurs if **d** is negative.

double ceil (double d)
float ceilf (float d)
long double ceill (long double d) Rounding Up

Description: Returns the smallest integral value not less than or equal to the given floating-point number.

Header file: <math.h>

Return values: Smallest integral value not less than or equal to **d**

Parameters:

d	Floating-point number for which smallest integral value not less than that number is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=ceil(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.

double fabs (double d)
float fabsf (float d)
long double fabsl (long double d) Absolute Value

Description: Calculates the absolute value of a floating-point number.

Header file: <math.h>

Return values: Absolute value of **d**

Parameters: d Floating-point number for which absolute value is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=fabs(d);
```

double floor (double d)
float floorf (float d)
long double floorl (long double d) Truncation

Description: Returns the largest integral value not greater than or equal to the given floating-point number.

Header file: <math.h>

Return values: Largest integral value not greater than or equal to **d**

Parameters:

d	Floating-point number for which largest integral value not greater than that number is to be computed

Example:

```
#include <math.h>
double d, ret;
    ret=floor(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.

double fmod (double x, double y)
float fmodf (float x, float y)
long double fmodl (long double x, long double y) Remainder

Description: Calculates the remainder of a division of two floating-point numbers.

Header file: <math.h>

Return values: When **y** is 0.0: **x**
When **y** is not 0.0: Remainder of division of **x** by **y**

Parameters:

x	Dividend
y	Divisor

Example:

```
#include <math.h>
double x, y, ret;
    ret=fmod(x, y);
```

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.

double acosh(double d)
float acoshf(float d)
long double acoshl(long double d) Hyperbolic Arc Cosine

Description: Calculates the hyperbolic arc cosine of a floating-point number.

Header file: <math.h>

Return values: Normal: Hyperbolic arc cosine of **d**
Abnormal: Domain error: Returns **NaN**.

Parameters: d Floating-point number for which hyperbolic arc cosine is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=acosh(d);
```

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, +∞].

double asinh(double d)
float asinhf(float d)
long double asinhl(long double d) Hyperbolic Arc Sine

Description: Calculates the hyperbolic arc sine of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic arc sine of **d**

Parameters: d Floating-point number for which hyperbolic arc sine is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=asinh(d);
```

double atanh(double d)
float atanhf(float d)
long double atanhl(long double d) Hyperbolic Arc Tangent

Description: Calculates the hyperbolic arc tangent of a floating-point number.

Header file: <math.h>

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.

Parameters:
d	Floating-point number for which hyperbolic arc tangent is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=atanh(d);
```

Error conditions: 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.

double exp2(double d)
float exp2f(float d)
long double exp2l(long double d) Exponential Function

Description: Calculates the value of 2 raised to the power **d**.

Header file: <math.h>

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

Parameters:

d	Floating-point number for which exponential function is to be computed

Example:

```
#include <math.h>
double d, ret;
    ret=exp2(d);
```

Error conditions: A range error occurs if the absolute value of **d** is too large.

double expm1(double d)
float expm1f(float d)
long double expm1l(long double d) Logarithm

Description: Calculates the value of natural logarithm base **e** raised to the power **d** and subtracts 1 from the result.

Header file: <math.h>

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.

Parameters:
- d: Power value to which natural logarithm base **e** is to be raised

Example:

```
#include <math.h>
double d, ret;
    ret=expm1(d);
```

Error conditions: A range error occurs if **d** is too large.

Remarks: **expm1(d)** provides more accurate calculation than **exp(x) – 1** even when **d** is near to 0.

long ilogb(double d)
long ilogbf(float d)
long ilogbl(long double d) Extracting Exponent

Description: Extracts the exponent of **d**.

Header file: <math.h>

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**

Parameters:

d	Value of which exponent is to be extracted

Example:

```
#include <math.h>
double d;
int ret;
    ret = ilogb(d);
```

Error conditions: A range error may occur if **d** is 0.

double log1p(double d)
float log1pf(float d)
long double log1pl(long double d) Logarithm

Description: Calculates the natural logarithm (base e) of **d** + 1.

Header file: <math.h>

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.

Parameters:
d	Value for which the natural logarithm of this parameter + 1 is to be computed

Example:
```
#include <math.h>
double d;
double ret;
    ret = log1p(d);
```

Error conditions: A domain error occurs if **d** is smaller than –1.
A range error occurs if **d** is –1.

Remarks: **log1p(d)** provides more accurate calculation than **log(1+d)** even when **d** is near to 0.

double log2(double d)
float log2f(float d)
long double log2l(long double d) Logarithm

Description: Calculates the base-2 logarithm of **d**.

Header file: <math.h>

Return values:

Normal:	Base-2 logarithm of **d**
Abnormal:	Domain error: Returns not-a-number.

Parameters:

d	Value of which logarithm is to be calculated

Example:

```
#include <math.h>
double d;
int ret;
    ret = log2(d);
```

Error conditions: A domain error occurs if **d** is a negative value.

double logb(double d)
float logbf(float d)
long double logbl(long double d) Extracting Exponent

Description: Extracts the exponent of **d** in internal floating-point representation, as a floating-point value.

Header file: <math.h>

Return values:
Normal: Signed exponent of **d**
Abnormal: Range error: Returns –**HUGE_VAL**, –**HUGE_VALF**, or –**HUGE_VALL** depending on the function.

Parameters:
d Value of which exponent is to be extracted

Example:
```
#include <math.h>
double d, ret;
    ret = logb(d);
```

Error conditions: A range error may occur if **d** is 0.

Remarks: **d** is always assumed to be normalized.

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)

Multiplication between Floating-Point Number and FLT_RADIX

Description: Calculates a floating-point number multiplied by a power of radix, which is an integer.

Header file: <math.h>

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.

Parameters:
d Value to be multiplied by **FLT_RADIX** raised to the power **e**
e Exponent used to compute a power of **FLT_RADIX**

Example:
```
#include <math.h>
double d, ret;
int e;
    ret = scalbn(d,e);
```

Error conditions: A range error may occur if **d** is 0.

Remarks: **FLT_RADIX** raised to the power **e** is not actually calculated.

double cbrt(double d)
float cbrtf(float d)
long double cbrtl(long double d) Cube Root

Description: Calculates the cube root of a floating-point number.

Header file: <math.h>

Return values: Cube root of **d**

Parameters:

d	Value for which a cube root is to be computed

Example:

```
#include <math.h>
double d, ret;
    ret = cbrt(d);
```

double hypot(double d, double e)
float hypotf(float d, double e)
long double hypotl(long double d, double e) Euclidean Distance

Description: Calculates the square root of the sum of floating-point numbers raised to the power 2.

Header file: <math.h>

Return values:

Normal:	Square root function value of sum of **d** raised to the power 2 and **e** raised to the power 2
Abnormal:	Range error: Returns **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** depending on the function.

Parameters:

d	Values for which the square root of the sum of these values raised to the power 2 is to be computed
e	

Example:

```
#include <math.h>
double d, e, ret;
    ret = hypot(d, e);
```

Error conditions: A range error may occur if the result overflows.

double erf(double d)
float erff(float d)
long double erfl(long double d) Error

Description: Calculates the error function value of a floating-point number.

Header file: <math.h>

Return values: Error function value of **d**

Parameters: d Value for which the error function value is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret = erf(d);
```

double erfc(double d)
float erfcf(float d)
long double erfcl(long double d) Complementary Error

Description: Calculates the complementary error function value of a floating-point number.

Header file: <math.h>

Return values: Complementary error function value of **d**

Parameters: d Value for which the complementary error function value is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret = erfc(d);
```

Error conditions: A range error occurs if the absolute value of **d** is too large.

double lgamma(double d)
float lgammaf(float d)
long double lgammal(long double d) Logarithm of Gamma Function

Description: Calculates the logarithm of the gamma function of a floating-point number.

Header file: <math.h>

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**.

Parameters:
d: Value for which the logarithm of the gamma function is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret = lgamma(d);
```

Error conditions: 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.

double tgamma(double d)
float tgammaf(float d)
long double tgammal(long double d) Gamma

Description: Calculates the gamma function of a floating-point number.

Header file: <math.h>

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.

Parameters:
d Value for which the gamma function value is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret = tgamma(d);
```

Error conditions: 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.

double nearbyint(double d)
float nearbyintf(float d)
long double nearbyintl(long double d) Conversion to Integer

Description: Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.

Header file: <math.h>

Return values: **d** rounded to an integer in the floating-point format

Parameters: d Value to be rounded to an integer in the floating-point format

Example:
```
#include <math.h>
double d, ret;
    ret = nearbyint(d);
```

Remarks: The **nearbyint** function group does not generate "inexact" floating-point exceptions.

double rint(double d)
float rintf(float d)
long double rintl(long double d) Conversion to Integer

Description: Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.

Header file: <math.h>

Return values: **d** rounded to an integer in the floating-point format

Parameters: d Value to be rounded to an integer in the floating-point format

Example:
```
#include <math.h>
double d, ret;
    ret = rint(d);
```

Remarks: The **rint** function group differs from the **nearbyint** function group only in that the **ring** function group may generate "inexact" floating-point exceptions.

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) Conversion to Integer

Description: Rounds a floating-point number to the nearest integer according to the current rounding direction.

Header file: <math.h>

Return values: Normal: **d** rounded to an integer
Abnormal: Range error: Returns an undetermined value.

Parameters: d Value to be rounded to an integer

Example:
```
#include <math.h>
double d;
long int ret;
    ret = lrint(d);
```

Error conditions: A range error may occur if the absolute value of **d** is too large.

Remarks: The return value is unspecified when the rounded value is not in the range of the return value type.

double round(double d)
float roundf(float d)
long double roundl(long double d)
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) Conversion to Integer

Description: Rounds a floating-point number to the nearest integer.

Header file: <math.h>

Return values: Normal: **d** rounded to an integer
Abnormal: Range error: Returns an undetermined value.

Parameters: d Value to be rounded to an integer

Example:
```
#include <math.h>
double d;
long int ret;
    ret = lround(d);
```

Error conditions: A range error may occur if the absolute value of **d** is too large.

Remarks: 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.

double trunc(double d)
float truncf(float d)
long double truncl(long double d) Conversion to Integer

Description: Rounds a floating-point number to the nearest integer in the floating-point representation.

Header file: <math.h>

Return values: **d** truncated to an integer in the floating-point format

Parameters:

d	Value to be rounded to an integer in the floating-point representation

Example:

```
#include <math.h>
double d, ret;
    ret = trunc(d);
```

Remarks: The **trunc** function group rounds **d** so that the absolute value after rounding is not greater than the absolute value of **d**.

double remainder(double d1, double d2)
float remainderf(float d1, float d2) Floating-Point
long double remainderl(long double d1, long double d2) Remainder Calculation

Description: Calculates the remainder of a division of two floating-point numbers.

Header file: <math.h>

Return values: Remainder of division of **d1** by **d2**

Parameters:

d1	Values for which remainder of a division is to be computed
d2	

Example:

```
#include <math.h>
double d1, d2, ret;
    ret = remainder(d1, d2);
```

Remarks: The remainder calculation by the **remainder** function group conforms to the IEEE 60559 standard.

double remquo(double d1, double d2, long *q)
float remquof(float d1, float d2, long *q) Floating-Point
long double remquol(long double d1, long double d2, long *q) Remainder Calculation

Description: Calculates the remainder of a division of two floating-point numbers.

Header file: <math.h>

Return values: Remainder of division of **d1** by **d2**

Parameters:

d1	Values for which remainder of a division is to be computed
d2	
q	Value pointing to the location to store the quotient obtained by remainder calculation

Example:

```
#include <math.h>
double d1, d2, ret;
long q;
    ret = remquo(d1, d2, &q);
```

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).

double copysign(double d1, double d2)
float copysignf(float d1, float d2)
long double copysignl(long double d1, long double d2) Sign Copy

Description: Generates a value consisting of the absolute value of **d1** and the sign of **d2**.

Header file: <math.h>

Return values:

Normal:	Value consisting of absolute value of **d1** and sign of **d2**
Abnormal:	Range error: Returns an undetermined value.

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

Example:

```
#include <math.h>
double d1, d2, ret;
    ret = copysign(d1, d2);
```

Remarks: When **d1** is a not-a-number, the **copysign** function group generates a not-a-number with the sign bit of **d2**.

double nan(const char *c)
float nanf(const char *c)
long double nanl(const char *c) Not-a-Number

Description: Returns not-a-number.

Header file: <math.h>

Return values: **qNaN** with the contents of the location indicated by **c** or 0 (when **qNaN** is not supported)

Parameters: c Pointer to a string

Example:
```
#include <math.h>
double ret;
const char *c;
    ret = nan(c);
```

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.

double nextafter(double d1, double d2)
float nextafter f(float d1, float d2)
long double nextafterl(long double d1, long double d2) Floating-Point Manipulation

Description: Calculates the next floating-point representation following **d1** in the direction to **d2** on the real axis.

Header file: <math.h>

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.

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

Example:

```
#include <math.h>
double d1, d2, ret;
    ret = nextafter(d1, d2);
```

Error conditions: 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.

Remarks: The **nextafter** function group returns **d2** when **d1** is equal to **d2**.

double nexttoward(double d1, long double d2)
float nexttowardf(float d1, long double d2)
long double nexttowardl(long double d1, long double d2) Floating-Point Manipulation

Description: Calculates the next floating-point representation following **d1** in the direction to **d2** on the real axis.

Header file: <math.h>

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

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

Example:

```
#include <math.h>
double d1, ret;
long double d2;
    ret = nexttoward(d1, d2);
```

Error conditions: 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.

Remarks: 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**.

double fdim(double d1, double d2)
float fdimf(float d1, float d2)
long double fdiml(long double d1, long double d2) Positive Difference

Description: Calculates the positive difference between two arguments.

Header file: <math.h>

Return values: Normal: Positive difference between two arguments
Abnormal: Range error: **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**

Parameters: d1 Values of which difference is to be computed
d2

Example:
```
#include <math.h>
double d1, d2, ret;
    ret = fdim(d1, d2);
```

Error conditions: A range error may occur if the return value overflows.

double fmax(double d1, double d2)
float fmaxf(float d1, float d2)
long double fmaxl(long double d1, long double d2) Maximum Value

Description: Obtains the greater of two arguments.

Header file: <math.h>

Return values: Greater of two arguments

Parameters: d1 Values to be compared
d2

Example:
```
#include <math.h>
double d1, d2, ret;
    ret = fmax(d1, d2);
```

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.

double fmin(double d1, double d2)
float fminf(float d1, float d2)
long double fminl(long double d1, long double d2) Minimum Value

Description: Obtains the smaller of two arguments.

Header file: <math.h>

Return values: Smaller of two arguments

Parameters: d1 Values to be compared
d2

Example:
```
#include <math.h>
double d1, d2, ret;
    ret = fmin(d1, d2);
```

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.

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) Multiply and Add

Description: Calculates (**d1** * **d2**) + **d3** as a single ternary operation.

Header file: <math.h>

Return values: Result of (**d1** * **d2**) + **d3** calculated as ternary operation

Parameters: d1, d2, d3 Floating-point values

Example:
```
#include <math.h>
double d1, d2, ret;
    ret = fma(d1, d2);
```

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**.

(9) <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	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.

Type	Definition Name	Description
Function	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.

Example:

```
      .
      .
      .
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.

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 9.1.3, 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.

float acosf (float f) Arc Cosine

Description: Calculates the arc cosine of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Arc cosine of **f**
Abnormal: Domain error: Returns not-a-number.

Parameters: f Floating-point number for which arc cosine is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=acosf(f);
```

Error conditions: A domain error occurs for a value of **f** not in the range [–1.0, +1.0].

Remarks: The **acosf** function returns the arc cosine in the range [0, π] by the radian.

float asinf (float f) Arc Sine

Description: Calculates the arc sine of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Arc sine of **f**
Abnormal: Domain error: Returns not-a-number.

Parameters: f Floating-point number for which arc sine is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=asinf(f);
```

Error conditions: A domain error occurs for a value of **f** not in the range [–1.0, +1.0].

Remarks: The **asinf** function returns the arc sine in the range [–π/2, +π/2] by the radian.

float atanf (float f) Arc Tangent

Description: Calculates the arc tangent of a floating-point number.

Header file: <mathf.h>

Return values: Arc tangent of **f**

Parameters: f Floating-point number for which arc tangent is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=atanf(f);
```

Remarks: The **atanf** function returns the arc tangent in the range (–π/2, +π/2) by the radian.

float atan2f (float y, float x) Arc Tangent after Division

Description: Calculates the arc tangent of the division of two floating-point numbers.

Header file: <mathf.h>

Return values:
Normal:	Arc tangent value when **y** is divided by **x**
Abnormal:	Domain error: Returns not-a-number.

Parameters:
x	Divisor
y	Dividend

Example:
```
#include <mathf.h>
float x, y, ret;
    ret=atan2f(y, x);
```

Error conditions: A domain error occurs if the values of both **x** and **y** are 0.0.

Remarks: The **atan2f** function returns the arc tangent in the range (–π, +π) by the radian. The meaning of the **atan2f** function is illustrated in figure 9.5. 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 ±π/2, depending on whether **y** is positive or negative.

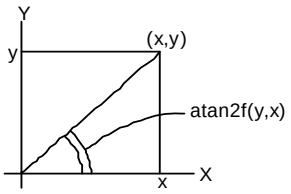


Figure 9.5 Meaning of atan2f Function

float cosf (float f) Cosine

Description: Calculates the cosine of a floating-point radian value.

Header file: <mathf.h>

Return values: Cosine of **f**

Parameters: f Radian value for which cosine is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=cosf(f);
```

float sinf (float f) Sine

Description: Calculates the sine of a floating-point radian value.

Header file: <mathf.h>

Return values: Sine of **f**

Parameters: f Radian value for which sine is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=sinf(f);
```

float tanf (float f) Tangent

Description: Calculates the tangent of a floating-point radian value.

Header file: <mathf.h>

Return values: Tangent of **f**

Parameters: f Radian value for which tangent is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=tanf(f);
```

float coshf (float f) Hyperbolic Cosine

Description: Calculates the hyperbolic cosine of a floating-point number.

Header file: <mathf.h>

Return values: Hyperbolic cosine of **f**

Parameters:

f	Floating-point number for which hyperbolic cosine is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=coshf(f);
```

float sinhf (float f) Hyperbolic Sine

Description: Calculates the hyperbolic sine of a floating-point number.

Header file: <mathf.h>

Return values: Hyperbolic sine of **f**

Parameters:

f	Floating-point number for which hyperbolic sine is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=sinhf(f);
```

float tanhf (float f) Hyperbolic Tangent

Description: Calculates the hyperbolic tangent of a floating-point number.

Header file: <mathf.h>

Return values: Hyperbolic tangent of **f**

Parameters:

f	Floating-point number for which hyperbolic tangent is to be computed

Example:

```
#include <mathf.h>
float f, ret;
    ret=tanhf(f);
```

float expf (float f) Exponential Function

Description: Calculates the exponential function of a floating-point number.

Header file: <mathf.h>

Return values: Exponential function value of **f**

Parameters:

f	Floating-point number for which exponential function is to be computed

Example:

```
#include <mathf.h>
float f, ret;
    ret=expf(f);
```

float frexpf (float value, float long *exp)

Breaking Floating-Point Number into Mantissa and Exponent

Description: Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.

Header file: <mathf.h>

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

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

Example:

```
#include <mathf.h>
float ret, value;
long *exp
    ret=frexpf(value, exp);
```

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.

float ldexpf (float e, long f) Converting Mantissa and Exponent into Floating-Point Number

Description: Multiplies a floating-point number by a power of 2.

Header file: <mathf.h>

Return values: Result of e * 2^f operation

Parameters:

e	Floating-point number to be multiplied by a power of 2
f	Power-of-2 value

Example:

```
#include <mathf.h>
float ret, e;
int f;
    ret=ldexpf(e, f);
```

float logf (float f) Natural Logarithm

Description: Calculates the natural logarithm of a floating-point number.

Header file: <mathf.h>

Return values:

Normal:	Natural logarithm of **f**
Abnormal:	Domain error: Returns not-a-number.

Parameters:

f	Floating-point number for which natural logarithm is to be computed

Example:

```
#include <mathf.h>
float f, ret;
    ret=logf(f);
```

Error conditions: A domain error occurs if **f** is negative.

A range error occurs if **f** is 0.0.

float log10f (float f) Base-Ten Logarithm

Description: Calculates the base-ten logarithm of a floating-point number.

Header file: <mathf.h>

Return values:
Normal:	Base-ten logarithm of **f**
Abnormal:	Domain error: Returns not-a-number.

Parameters:
f	Floating-point number for which base-ten logarithm is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=log10f(f);
```

Error conditions: A domain error occurs if **f** is negative.

A range error occurs if **f** is 0.0.

float modff (float a, float *b) Breaking Floating-Point Number into Integral and Fractional Parts

Description: Breaks a floating-point number into integral and fractional parts.

Header file: <mathf.h>

Return values: Fractional part of **a**

Parameters:
a	Floating-point number to be broken into integral and fractional parts
b	Pointer indicating storage area that stores integral part

Example:
```
#include <mathf.h>
float a, *b, ret;
    ret=modff(a, b);
```

float powf (float x, float y) Power of Floating-Point Number

Description: Calculates a power of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Value of **x** raised to the power **y**
Abnormal: Domain error: Returns not-a-number.

Parameters: x Value to be raised to a power
y Power value

Example:
```
#include <mathf.h>
float x, y, ret;
    ret=powf(x, y);
```

Error conditions: 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.

float sqrtf (float f) Square Root

Description: Calculates the positive square root of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Positive square root of **f**
Abnormal: Domain error: Returns not-a-number.

Parameters: f Floating-point number for which positive square root is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=sqrtf(x, y);
```

Error conditions: A domain error occurs if **f** is negative.

float ceilf (float f) Rounding Up

Description: Returns the smallest integral value not less than or equal to the given floating-point number.

Header file: <mathf.h>

Return values: Smallest integral value not less than or equal to **f**

Parameters:

f	Floating-point number for which smallest integral value not less than that number is to be computed

Example:

```
#include <mathf.h>
float f, ret;
    ret=ceilf(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.

float fabsf (float f) Absolute Value

Description: Calculates the absolute value of a floating-point number.

Header file: <mathf.h>

Return values: Absolute value of **f**

Parameters:

f	Floating-point number for which absolute value is to be computed

Example:

```
#include <mathf.h>
float f, ret;
    ret=fabsf(f);
```

float floorf (float f) Truncation

Description: Returns the largest integral value not greater than or equal to the given floating-point number.

Header file: <mathf.h>

Return values: Largest integral value not greater than or equal to **f**

Parameters:

f	Floating-point number for which largest integral value not greater than that number is to be computed

Example:

```
#include <mathf.h>
float f, ret;
    ret=floorf(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.

float fmodf (float x, float y) Remainder

Description: Calculates the remainder of a division of two floating-point numbers.

Header file: <mathf.h>

Return values: When **y** is 0.0: **x**
When **y** is not 0.0: Remainder of division of **x** by **y**

Parameters:

x	Dividend
y	Divisor

Example:

```
#include <mathf.h>
float x, y, ret;
    ret=fmodf(x, 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.

(10) <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.

Example:

```
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.

long setjmp (jmp_buf env) Global goto Destination Setting

Description: Saves the execution environment of the currently executing function in the specified storage area.

Header file: <setjmp.h>

Return values: When **setjmp** function is called: 0
On return from **longjmp** function: Nonzero

Parameters:

env	Pointer to storage area in which execution environment is to be saved

Example:

```
#include <setjmp.h>
int ret;
jmp_buf env;
    ret=setjmp(env);
```

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.

void longjmp (jmp_buf env, long ret) Global goto

Description: Restores the function execution environment saved by the **setjmp** function, and transfers control to the program location at which the **setjmp** function was called.

Header file: <setjmp.h>

Parameters:

env	Pointer to storage area in which execution environment was saved
ret	Return code to **setjmp** function

Example:

```
#include <setjmp.h>
int ret;
jmp_buf env;
    longjmp(env, ret);
```

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.

(11) <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	Copies variable arguments.

An example of a program using the macros defined by this standard include file is shown below.

Example:

```
 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).

void va_start (va_list ap, parmN) Variable Argument Referencing Start

Description:	Executes initialization processing for referencing variable arguments.	
Header file:	<stdarg.h>	
Parameters:	ap	Variable for accessing variable arguments
	parmN	Identifier of rightmost argument

Example:

```
#include <stdarg.h>
void func(int count, ...)
{
   va_list ap;
     va_start(ap, count);
}
```

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.

type va_arg (va_list ap, type) Variable Argument Referencing

Description: Allows a reference to the argument following the argument currently being referred to in the function with variable arguments.

Header file: <stdarg.h>

Return values: Argument value

Parameters:

ap	Variable for accessing variable arguments
type	Type of arguments to be accessed

Example:

```
#include <stdarg.h>
va_list ap;
int ret;
    ret=va_arg(ap, type);
```

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.

void va_end (va_list ap) Variable Argument Referencing End

Description: Terminates referencing of the arguments of a function with variable arguments.

Header file: <stdarg.h>

Parameters: ap Variable for referencing variable arguments

Example:
```
#include <stdarg.h>
va_list ap;
    va_end(ap);
```

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.

void va_copy (va_list dest, va_list src) Variable Argument Copy

Description: Makes a copy of the argument currently being referenced for a function with variable arguments.

Header file: <stdarg.h>

Parameters: dest Copy of variable for referencing variable arguments
src Variable for referencing variable arguments

Example:
```
#include <stdarg.h>
va_list ap, ap_sub;
    va_copy(ap_sub,ap);
```

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.

(12) <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.
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.

Type	Definition Name	Description
Function	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	Converts data according to a format and writes it to the specified array.
	vsnprintf	Equivalent to **snprintf** with the variable argument list replaced by **va_list**.
	vfscanf	Equivalent to **fscanf** with the variable argument list replaced by **va_list**.
	vscanf	Equivalent to **scanf** with the variable argument list replaced by **va_list**.
	vsscanf	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.

Type	Definition Name	Description
Function	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.
	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.
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 9.31 shows the format when displaying the floating-point infinity and not-a-number in **printf** and **fprintf** functions.

Table 9.31 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.

Example:

```
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.

long fclose (FILE *fp) File Close

Description: Closes a stream input/output file.

Header file: <stdio.h>

Return values: Normal: 0
Abnormal: Nonzero

Parameters: fp File pointer

Example:
```
#include <stdio.h>
FILE *fp;
int ret;
    ret=fclose(fp);
```

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.

long fflush (FILE *fp) Buffer Flush

Description: Outputs the stream input/output file buffer contents to the file.

Header file: <stdio.h>

Return values: Normal: 0
Abnormal: Nonzero

Parameters: fp File pointer

Example:
```
#include <stdio.h>
FILE *fp;
int ret;
    ret=fflush(fp);
```

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.

FILE *fopen (const char *fname, const char *mode) File Open

Description: Opens a stream input/output file under the specified file name.

Header file: <stdio.h>

Return values:
Normal:	File pointer indicating file information on opened file
Abnormal:	**NULL**

Parameters:
fname	Pointer to string indicating file name
mode	Pointer to string indicating file access mode

Example:
```
#include <stdio.h>
FILE *ret;
const char *fname, *mode;
    ret=fopen(fname, mode);
```

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.

FILE *freopen (const char *fname, const char *mode, FILE *fp) File Reopen

Description: Closes a currently open stream input/output file and reopens a new file under the specified file name.

Header file: <stdio.h>

Return values:
Normal: **fp**
Abnormal: **NULL**

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

Example:

```
#include <stdio.h>
const char *fname, *mode;
FILE *ret, *fp;
    ret=freopen(fname, mode, fp);
```

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.

void setbuf (FILE *fp, char buf[BUFSIZ]) Buffer Area Setting

Description: Defines and sets a stream input/output buffer area by the user program.

Header file: <stdio.h>

Parameters:

fp	File pointer
buf	Pointer to buffer area

Example:

```
#include <stdio.h>
FILE *fp;
char buf[BUFSIZ];
    setbuf(fp, buf);
```

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**.

long setvbuf (FILE *fp, char *buf, long type, size_t size) Buffer Control

Description: Defines and sets a stream input/output buffer area by the user program.

Header file: <stdio.h>

Return values:

Normal:	0
Abnormal:	Nonzero

Parameters:

fp	File pointer
buf	Pointer to buffer area
type	Buffer management method
size	Size of buffer area

Example:

```
#include <stdio.h>
FILE *fp;
char *buf;
int type, ret;
size_t size;
    ret=setvbuf(fp, buf, type, size);
```

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.

long fprintf (FILE *fp, const char *control[, arg…]) Formatted Output to File

Description: Outputs data to a stream input/output file according to the format.

Header file: <stdio.h>

Return values:

Normal:	Number of characters converted and output
Abnormal:	Negative value

Parameters:

fp	File pointer
control	Pointer to string indicating format
arg,…	List of data to be output according to format

Example:

```
#include <stdio.h>
FILE *fp;
const char *control="%s";
int ret;
char buffer[]="Hello World\n";
    ret=fprintf(fp, control, buffer);
```

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:

  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 9.32.

Table 9.32 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.
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 9.34. 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 9.34.

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 9.34), 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 9.33 shows the types of size specification and their meanings.

Table 9.33 Parameter Size Specification Types and Meanings

Type	Meaning
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.
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.
L	For **e**, **E**, **f**, **g**, and **G** conversions, specifies that the data to be converted is of **long double** type.
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.

(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 9.34 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 9.34 Conversion Specifiers and Conversion Methods

Conversion Specifier	Conversion Type	Conversion Method	Data Type Subject to Conversion	Notes on Precision
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.
i	i conversion		**int** type	
o	o conversion	**int** type data is converted to an unsigned octal string.	**int** type	
u	u conversion	**int** type data is converted to an unsigned decimal string.	**int** type	
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	
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	
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.
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.
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	

Conversion Specifier	Conversion Type	Conversion Method	Data Type Subject to Conversion	Notes on Precision
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.
G			**double** type	
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.
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.)
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.
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	
%	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.

long snprintf(char *restrict s, size_t n,
const char *restrict control [, arg] …) Formatted String Output

Description: Converts data according to a format and outputs it to the specified area.

Header file: <stdio.h>

Return values: Number of characters converted

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

Example:

```
#include <stdio.h>
char *s;
size_t n;
const char *control="%s";
int ret;
char buffer[]="Hello World\n";
ret=snprintf(s,n,control,buffer);
```

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.

long vsnprintf(char *restrict s, size_t n, const char *restrict control, va_list arg) Variable-Parameter Formatted String Output

Description: Converts data according to a format and outputs it to the specified area.

Header file: <stdarg.h>, <stdio.h>

Return values: Number of characters converted

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

Example:

```
#include <stdarg.h>
#include <stdio.h>
char *s;
size_t n;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
    va_list ap;
    int i;
    va_start(ap, count);
    for(i=0;i<count;i++) {
       ret=vsnprintf(s,control,ap);
       va_arg(ap,int);
       s += ret;
    }
}
```

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.

long fscanf (FILE *fp, const char *control[, ptr…]) Formatted Input from File

Description: Inputs data from a stream input/output file and converts it according to a format.

Header file: <stdio.h>

Return values:

Normal:	Number of data items successfully input and converted
Abnormal:	Input data ends before input data conversion is performed: **EOF**

Parameters:

fp	File pointer
control	Pointer to string indicating format
ptr,...	Pointer to storage area that stores input data

Example:

```
#include <stdio.h>
FILE *fp;
const char *control="%d";
int ret,buffer[10];
    ret=fscanf(fp, control, buffer);
```

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 9.36), 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 9.35 shows the types of size specification and their meanings.

Table 9.35 Converted Data Size Specification Types and Meanings

Type	Meaning
h	For **d**, **i**, **o**, **u**, **x**, and **X** conversions, specifies that the converted data is of **short** type.
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.
L	For **e**, **E**, and **f** conversions, specifies that the converted data is of **long double** type.
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 9.36 Conversion Specifiers and Conversion Methods

Conversion Specifier	Conversion Type	Conversion Method	Data Type Subject to Conversion
d	d conversion	A decimal string is converted to integer type data.	Integer type
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
o	o conversion	An octal string is converted to integer type data.	Integer type
u	u conversion	An unsigned decimal string is converted to integer type data.	Integer type
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
X	X conversion		
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
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
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
E	E conversion		
f	f conversion		
g	g conversion		
G	G conversion		
p	p conversion	A string converted by **p** conversion of the **fprintf** function is converted to pointer type data.	Pointer to **void** type
n	No conversion is performed.	Data input is not performed; the number of data characters input so far is set.	Integer type
[	[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
%	No conversion is performed.	% is read.	None

If the conversion specifier is a letter not shown in table 9.36, the behavior is not guaranteed. For the other characters, the behavior is implementation-defined.

long printf (const char *control[, arg…]) Formatted Output

Description: Converts data according to a format and outputs it to the standard output file (**stdout**).

Header file: <stdio.h>

Return values:
Normal:	Number of characters converted and output
Abnormal:	Negative value

Parameters:
control	Pointer to string indicating format
arg,...	Data to be output according to format

Example:
```
#include <stdio.h>
const char *control="%s";
int ret;
char buffer[]="Hello World\n";
    ret=printf(control, buffer);
```

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.

long vfscanf(FILE *restrict fp, const char *restrict control, va_list arg) Variable-Parameter Formatted Input from File

Description: Inputs data from a stream input/output file and converts it according to a format.

Header file: <stdarg.h>, <stdio.h>

Return values:

Normal:	Number of data items successfully input and converted
Abnormal:	Input data ends before input data conversion is performed: **EOF**

Parameters:

fp	File pointer
control	Pointer to wide string indicating format
arg	Parameter list

Example:

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

FILE *fp;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
    va_list ap;
    int i;
    va_start(ap, count);
    for(i=0;i<count;i++)
       ret=vfscanf(fp, control, ap);
    va_end(ap);
}
```

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.

long scanf (const char *control[, ptr…]) Formatted Input

Description: Inputs data from the standard input file (**stdin**) and converts it according to a format.

Header file: <stdio.h>

Return values:
Normal: Number of data items successfully input and converted
Abnormal: **EOF**

Parameters:
control: Pointer to string indicating format
ptr,...: Pointer to storage area that stores input and converted data

Example:
```
#include <stdio.h>
const char *control="%d";
int ret,buffer[10];
    ret=scanf(control,buffer);
```

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**.

**long vscanf(const char *restrict control,
va_list arg)** Variable-Parameter Formatted Input from File

Description: Inputs data from the specified storage area and converts it according to a format.

Header file: <stdarg.h>, <stdio.h>

Return values:

Normal:	Number of data items successfully input and converted
Abnormal:	Input data ends before input data conversion is performed: **EOF**

Parameters:

control	Pointer to string indicating format
arg	Parameter list

Example:

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

FILE *fp;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
    va_list ap;
    int i;
    va_start(ap, count);
    for(i=0;i<count;i++)
       ret=vscanf(control, ap);
    va_end(ap);
}
```

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.

long sprintf (char *s, const char *control[, arg…]) Formatted String Output

Description: Converts data according to a format and outputs it to the specified area.

Header file: <stdio.h>

Return values: Number of characters converted

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

Example:

```
#include <stdio.h>
char *s;
const char *control="%s";
int ret;
char buffer[]="Hello World\n";
    ret=sprintf(s, control, buffer);
```

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.

long sscanf (const char *s, const char *control[, ptr…]) Formatted String Input

Description: Inputs data from the specified storage area and converts it according to a format.

Header file: <stdio.h>

Return values:

Normal:	Number of data items successfully input and converted
Abnormal:	**EOF**

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

Example:

```
#include <stdio.h>
const char *s, *control="%d";
int ret,buffer[10];
    ret=sscanf(s, control, buffer);
```

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.

long vsscanf(const char *restrict s, const char *restrict control, va_list arg) Variable-Parameter Formatted Input from File

Description: Inputs data from the specified storage area and converts it according to a format.

Header file: <stdarg.h>, <stdio.h>

Return values:

Normal:	Number of data items successfully input and converted
Abnormal:	Input data ends before input data conversion is performed: **EOF**

Parameters:

s	Storage area containing data to be input
control	Pointer to string indicating format
arg	Parameter list

Example:

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

const char *s, *control="%d";
int ret;

void prlist(int count ,...)
{
    va_list ap;
    int i;
    va_start(ap, count);
    for(i=0;i<count;i++)
       ret=vsscanf(control, ap);
    va_end(ap);
}
```

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.

long vfprintf (FILE *fp, const char *control, va_list arg) Variable Parameter Output to File

Description: Outputs a variable parameter list to the specified stream input/output file according to a format.

Header file: <stdio.h>

Return values:

Normal:	Number of characters converted and output
Abnormal:	Negative value

Parameters:

fp	File pointer
control	Pointer to string indicating format
arg	Parameter list

Example:

```
#include <stdarg.h>
#include <stdio.h>
FILE *fp;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vfprintf(fp, control, ap);
   va_end(ap);
}
```

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).

long vprintf (const char *control, va_list arg) Variable Parameter Output

Description: Outputs a variable parameter list to the standard output file (**stdout**) according to a format.

Header file: <stdio.h>

Return values:

Normal:	Number of characters converted and output
Abnormal:	Negative value

Parameters:

control	Pointer to string indicating format
arg	Parameter list

Example:

```
#include <stdarg.h>
#include <stdio.h>
FILE *fp;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vprintf(control, ap);
   va_end(ap);
}
```

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).

long vsprintf (char *s, const char *control, va_list arg) Variable Parameter String Output

Description: Outputs a variable parameter list to the specified storage area according to a format.

Header file: <stdio.h>

Return values:

Normal:	Number of characters converted
Abnormal:	Negative value

Parameters:

s	Pointer to storage area to which data is to be output
control	Pointer to string indicating format
arg	Parameter list

Example:

```
#include <stdarg.h>
#include <stdio.h>
char *s;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++) {
      ret=vsprintf(s,control,ap);
      va_arg(ap,int)
      s += ret;
   }
}
```

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).

long fgetc (FILE *fp) — One Character Input from File

Description: Inputs one character from a stream input/output file.

Header file: <stdio.h>

Return values:

Normal:	End-of-file: **EOF**
	Otherwise: Input character
Abnormal:	**EOF**

Parameters:

fp	File pointer

Example:

```
#include <stdio.h>
FILE *fp;
int ret;
    ret=fgetc(fp);
```

Error conditions: When a read error occurs, the error indicator for that file is set.

Remarks: 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.

char *fgets (char *s, long n, FILE *fp) String Input from File

Description: Inputs a string from a stream input/output file.

Header file: <stdio.h>

Return values:

Normal:	End-of-file: **NULL**
	Otherwise: **s**
Abnormal:	**NULL**

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

Example:

```
#include <stdio.h>
char *s, *ret;
int n;
FILE *fp;
    ret=fgets(s, n, fp);
```

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.

long fputc (long c, FILE *fp) One Character Output to File

Description: Outputs one character to a stream input/output file.

Header file: <stdio.h>

Return values:

Normal:	Output character
Abnormal:	**EOF**

Parameters:

c	Character to be output
fp	File pointer

Example:

```
#include <stdio.h>
FILE *fp;
int c, ret;
    ret=fputc(c, fp);
```

Error conditions: When a write error occurs, the error indicator for that file is set.

Remarks: 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.

long fputs (const char *s, FILE *fp) String Output to File

Description: Outputs a string to a stream input/output file.

Header file: <stdio.h>

Return values:

Normal:	0
Abnormal:	Nonzero

Parameters:

s	Pointer to string to be output
fp	File pointer

Example:

```
#include <stdio.h>
const char *s;
int ret;
FILE *fp;
    ret=fputs(s, fp);
```

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.

long getc (FILE *fp) One Character Input from File

Description: Inputs one character from a stream input/output file.

Header file: <stdio.h>

Return values:
Normal:	End-of-file: **EOF**
	Otherwise: Input character
Abnormal:	**EOF**

Parameters:
fp	File pointer

Example:
```
#include <stdio.h>
FILE *fp;
int ret;
    ret=getc(fp);
```

Error conditions: When a read error occurs, the error indicator for that file is set.

Remarks: 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.

long getchar (void) One Character Input

Description: Inputs one character from the standard input file (**stdin**).

Header file: <stdio.h>

Return values:
Normal: End-of-file: **EOF**
Otherwise: Input character
Abnormal: **EOF**

Example:
```
#include <stdio.h>
int ret;
    ret=getchar();
```

Error conditions: When a read error occurs, the error indicator for that file is set.

Remarks: 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.

char *gets (char *s) String Input

Description: Inputs a string from the standard input file (**stdin**).

Header file: <stdio.h>

Return values:
- Normal: End-of-file: **NULL**; Otherwise: **s**
- Abnormal: **NULL**

Parameters:
- s: Pointer to storage area to which string is input

Example:

```
#include <stdio.h>
char *ret, *s;
    ret=gets(s);
```

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.

long putc (long c, FILE *fp) One Character Output to File

Description: Outputs one character to a stream input/output file.

Header file: <stdio.h>

Return values:
Normal:	Output character
Abnormal:	**EOF**

Parameters:
c	Character to be output
fp	File pointer

Example:
```
#include <stdio.h>
FILE *fp;
int c, ret;
    ret=putc(c, fp);
```

Error conditions: When a write error occurs, the error indicator for that file is set.

Remarks: 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.

long putchar (long c) One Character Output

Description: Outputs one character to the standard output file (**stdout**).

Header file: <stdio.h>

Return values: Normal: Output character
Abnormal: **EOF**

Parameters: c Character to be output

Example:
```
#include <stdio.h>
int c, ret;
    ret=putchar(c);
```

Error conditions: When a write error occurs, the error indicator for that file is set.

Remarks: 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.

long puts (const char *s) String Output

Description: Outputs a string to the standard output file (**stdout**).

Header file: <stdio.h>

Return values: Normal: 0
Abnormal: Nonzero

Parameters: s Pointer to string to be output

Example:
```
#include <stdio.h>
const char *s;
int ret;
    ret=puts(s);
```

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.

long ungetc (long c, FILE *fp) — One Character Return to File

Description: Returns one character to a stream input/output file.

Header file: <stdio.h>

Return values:

Normal:	Returned character
Abnormal:	**EOF**

Parameters:

c	Character to be returned
fp	File pointer

Example:

```
#include <stdio.h>
int c, ret;
FILE *fp;
    ret=ungetc(c, fp);
```

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.

size_t fread (void *ptr, size_t size, size_t n, FILE *fp) Reading from File

Description: Inputs data from a stream input/output file to the specified storage area.

Header file: <stdio.h>

Return values: When **size** or **n** is 0: 0
When **size** and **n** are both nonzero: Number of successfully input members

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

Example:

```
#include <stdio.h>
void *ptr;
size_t size;
size_t n, ret;
FILE *fp;
    ret=fread(ptr, size, n, fp);
```

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.

size_t fwrite (const void *ptr, size_t size, size_t n, FILE *fp) Writing to File

Description: Outputs data from a memory area to a stream input/output file.

Header file: <stdio.h>

Return values: Number of successfully output members

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

Example:

```
#include <stdio.h>
const void *ptr;
size_t size;
size_t n, ret;
FILE *fp;
    ret=fwrite(ptr, size, n, fp);
```

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.

long fseek (FILE *fp, long offset, long type) Shifting File Read/Write Position

Description: Shifts the current read/write position in a stream input/output file.

Header file: <stdio.h>

Return values:
Normal:	0
Abnormal:	Nonzero

Parameters:
fp	File pointer
offset	Offset from position specified by type of offset
type	Type of offset

Example:
```
#include <stdio.h>
FILE *fp;
long offset;
int type, ret;
    ret=fseek(fp, offset, type);
```

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 9.37.

The **fseek** function normally returns zero, but returns nonzero in response to an invalid request.

Table 9.37 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.

long ftell (FILE *fp) Obtaining File Read/Write Position

Description: Obtains the current read/write position in a stream input/output file.

Header file: <stdio.h>

Return values: Current file position indicator position (text file)
Number of bytes from beginning of file to current position (binary file)

Parameters: fp File pointer

Example:
```
#include <stdio.h>
FILE *fp;
long ret;
    ret=ftell(fp);
```

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.

void rewind (FILE *fp) Shifting to Beginning of File

Description: Shifts the current read/write position in a stream input/output file to the beginning of the file.

Header file: <stdio.h>

Parameters: fp File pointer

Example:
```
#include <stdio.h>
FILE *fp;
    rewind(fp);
```

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.

void clearerr (FILE *fp) Error State Clearing

Description: Clears the error state of a stream input/output file.

Header file: <stdio.h>

Parameters: fp File pointer

Example:
```
#include <stdio.h>
FILE *fp;
    clearerr(fp);
```

Remarks: The **clearerr** function clears the error indicator and end-of-file indicator for the stream input/output file indicated by file pointer **fp**.

long feof (FILE *fp) Test for End-of-File

Description: Tests for the end of a stream input/output file.

Header file: <stdio.h>

Return values: End-of-file: Nonzero
Otherwise: 0

Parameters: fp File pointer

Example:
```
#include <stdio.h>
FILE *fp;
int ret;
    ret=feof(fp);
```

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.

long ferror (FILE *fp) Test for File Error State

Description: Tests for stream input/output file error state.

Header file: <stdio.h>

Return values: If file is in error state: Nonzero
Otherwise: 0

Parameters: fp File pointer

Example:
```
#include <stdio.h>
FILE *fp;
int ret;
    ret=ferror(fp);
```

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.

void perror (const char *s) Error Message Output

Description: Outputs an error message corresponding to the error number to the standard error file (**stderr**).

Header file: <stdio.h>

Parameters: s Pointer to error message

Example:
```
#include <stdio.h>
const char *s;
    perror(s);
```

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.

(13) <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.

Type	Definition Name	Description
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.
	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.

double atof (const char *nptr) String Conversion to **double** Type

Description: Converts a number-representing string to a **double** type floating-point number.

Header file: <stdlib.h>

Return values: Converted data as a **double** type floating-point number

Parameters: nptr Pointer to a number-representing string to be converted

Example:
```
#include <stdlib.h>
const char *nptr;
double ret;
    ret=atof(nptr);
```

Error conditions: If the converted result overflows or underflows, **errno** is set.

Remarks: 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.

long atoi (const char *nptr) String Conversion to int Type

Description: Converts a decimal-representing string to an **int** type integer.

Header file: <stdlib.h>

Return values: Converted data as an **int** type integer

Parameters: nptr Pointer to a number-representing string to be converted

Example:
```
#include <stdlib.h>
const char *nptr;
int ret;
    ret=atoi(nptr);
```

Error conditions: If the converted result overflows, **errno** is set.

Remarks: 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.

long atol (const char *nptr) String Conversion to long Type

Description: Converts a decimal-representing string to a **long** type integer.

Header file: <stdlib.h>

Return values: Converted data as a **long** type integer

Parameters: nptr Pointer to a number-representing string to be converted

Example:
```
#include <stdlib.h>
const char *nptr;
long ret;
    ret=atol(nptr);
```

Error conditions: If the converted result overflows, **errno** is set.

Remarks: 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.

long long atoll (const char *nptr) String Conversion to long long Type

Description: Converts a decimal-representing string to a **long long** type integer.

Header file: <stdlib.h>

Return values: Converted data as a **long long** type integer

Parameters: nptr Pointer to a number-representing string to be converted

Example:
```
#include <stdlib.h>
const char *nptr;
long long ret;
    ret=atoll(nptr);
```

Error conditions: If the converted result overflows, **errno** is set.

Remarks: 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.

double strtod (const char *nptr, char **endptr) String Conversion to double Type

Description: Converts a number-representing string to a **double** type floating-point number.

Header file: <stdlib.h>

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

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

Example:

```
#include <stdlib.h>
const char *nptr;
char **endptr;
double ret;
    ret=strtod(nptr, endptr);
```

Error conditions: If the converted result overflows or underflows, **errno** is set.

Remarks: According to the rules described in section 9.1.3 (4), Floating-Point Operation Specifications, 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.

float strtof (const char *nptr, char **endptr) String Conversion to float Type

Description: Converts a number-representing string to a **float** type floating-point number.

Header file: <stdlib.h>

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

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

Example:

```
#include <stdlib.h>
const char *nptr;
char **endptr;
float ret;
    ret=strtof(nptr, endptr);
```

Error conditions: If the converted result overflows or underflows, **errno** is set.

Remarks: According to the rules described in section 9.1.3 (4), Floating-Point Operation Specifications, 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.

long double strtold (const char *nptr, char **endptr) String Conversion to long double Type

Description: Converts a number-representing string to a **long double** type floating-point number.

Header file: <stdlib.h>

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

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

Example:

```
#include <stdlib.h>
const char *nptr;
char **endptr;
long double ret;
    ret=strtold(nptr, endptr);
```

Error conditions: If the converted result overflows or underflows, **errno** is set.

Remarks: According to the rules described in section 9.1.3 (4), Floating-Point Operation Specifications, 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.

long strtol (const char *nptr, char **endptr, long base) String Conversion to long Type

Description: Converts a number-representing string to a **long** type integer.

Header file: <stdlib.h>

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

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)

Example:

```
#include <stdlib.h>
long ret;
const char *nptr;
char **endptr;
int base;
    ret=strtol(nptr, endptr, base);
```

Error conditions: If the converted result overflows, **errno** is set.

Remarks: 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 9.1.1 (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.

unsigned long strtoul (const char *nptr, char **endptr, long base)

String Conversion to unsigned long Type

Description: Converts a number-representing string to an **unsigned long** type integer.

Header file: <stdlib.h>

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**

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)

Example:

```
#include <stdlib.h>
unsigned long ret;
const char *nptr;
char **endptr;
int base;
    ret=strtoul(nptr, endptr, base);
```

Error conditions: If the converted result overflows, **errno** is set.

Remarks: 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 9.1.1 (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.

long long strtoll (const char *nptr, char **endptr, long base) — String Conversion to long long Type

Description: Converts a number-representing string to a **long long** type integer.

Header file: <stdlib.h>

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

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)

Example:

```
#include <stdlib.h>
long long ret;
const char *nptr;
char **endptr;
int base;
    ret=strtoll(nptr,endptr,base);
```

Error conditions: If the converted result overflows, **errno** is set.

Remarks: 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 9.1.1 (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.

unsigned long long strtoull (const char *nptr, char **endptr, long base) — String Conversion to unsigned long long Type

Description: Converts a number-representing string to an **unsigned long long** type integer.

Header file: <stdlib.h>

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**

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)

Example:

```
#include <stdlib.h>
unsigned long long ret;
const char *nptr;
char **endptr;
int base;
    ret=strtoull(nptr,endptr,base);
```

Error conditions: If the converted result overflows, **errno** is set.

Remarks: 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 9.1.1 (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.

long rand (void) Pseudo-Random Number Generation

Description: Generates a pseudo-random integer from 0 to **RAND_MAX**.

Header file: <stdlib.h>

Return values: Pseudo-random integer

Example:
```
#include <stdlib.h>
int ret;
    ret=rand();
```

void srand (unsigned long seed) Initial Setting for Pseudo-Random Number Sequence

Description: Sets an initial value of the pseudo-random number sequence generated by the **rand** function.

Header file: <stdlib.h>

Parameters: seed Initial value for pseudo-random number sequence generation

Example:
```
#include <stdlib.h>
unsigned int seed;
    srand(seed);
```

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.

void *calloc (size_t nelem, size_t elsize) — Storage Area Allocation and Initialization

Description: Allocates a storage area and clears all bits in the allocated storage area to 0.

Header file: <stdlib.h>

Return values:

Normal:	Starting address of an allocated storage area
Abnormal:	Storage allocation failed, or either of the parameter is 0: **NULL**

Parameters:

nelem	Number of elements
elsize	Number of bytes occupied by a single element

Example:
```
#include <stdlib.h>
size_t nelem, elsize;
void *ret;
    ret=calloc(nelem, elsize);
```

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.

void free (void *ptr) Storage Area Release

Description: Releases the specified storage area.

Header file: <stdlib.h>

Parameters: ptr Address of storage area to release

Example:
```
#include <stdlib.h>
void *ptr;
   free(ptr);
```

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.

void *malloc (size_t size) Storage Area Allocation

Description: Allocates a storage area.

Header file: <stdlib.h>

Return values: Normal: Starting address of allocated storage area
Abnormal: Storage allocation failed, or **size** is 0: **NULL**

Parameters: size Size in number of bytes of storage area to allocate

Example:
```
#include <stdlib.h>
size_t size;
void *ret;
    ret=malloc(size);
```

Remarks: The **malloc** function allocates a storage area of a specified number of bytes by **size**.

void *realloc (void *ptr, size_t size) Changing Allocated Storage Area Size

Description: Changes the size of a storage area to a specified value.

Header file: <stdlib.h>

Return values:

Normal:	Starting address of storage area whose size has been changed
Abnormal:	Storage area allocation has failed, or **size** is 0: **NULL**

Parameters:

ptr	Starting address of storage area to be changed
size	Size of storage area in number of bytes after the change

Example:

```
#include <stdlib.h>
size_t size;
void *ptr, *ret;
    ret=realloc(ptr, size);
```

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.

void *bsearch (const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *)) Binary Search

Description: Performs binary search.

Header file: <stdlib.h>

Return values: If a matching member is found: Pointer to the matching member
If no matching member is found: **NULL**

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

Example:

```
#include <stdlib.h>
const void *key, *base;
size_t nmemb, size;
int (*compar)(const void *, const void *);
void *ret;
    ret=bsearch(key, base, nmemb, size, compar);
```

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.

**void qsort (const void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *))** Sorting

Description: Performs sorting.

Header file: <stdlib.h>

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

Example:

```
#include <stdlib.h>
const void *base;
size_t nmemb, size;
int (*compar)(const void *, const void *)
    qsort(base, nmemb, size, compar);
```

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.

long abs (long i) Absolute Value

Description: Calculates the absolute value of an **int** type integer.

Header file: <stdlib.h>

Return values: Absolute value of **i**

Parameters:

i	Integer to calculate the absolute value of

Example:

```
#include <stdlib.h>
int i, ret;
    ret=abs(i);
```

Remarks: If the resultant absolute value cannot be expressed as an **int** type integer, correct operation is not guaranteed.

div_t div (long numer, long denom) Quotient and Remainder

Description: Carries out division of **int** type integers and obtains the quotient and remainder.

Header file: <stdlib.h>

Return values: Quotient and remainder of division of numer by denom

Parameters:

numer	Dividend
denom	Divisor

Example:

```
#include <stdlib.h>
int numer, denom;
div_t ret;
    ret=div(numer, denom);
```

long labs (long j) Absolute Value

Description: Calculates the absolute value of a **long** type integer.

Header file: <stdlib.h>

Return values: Absolute value of j

Parameters: j Integer to calculate the absolute value of

Example:
```
#include <stdlib.h>
long j;
long ret;
    ret=labs(j);
```

Remarks: If the resultant absolute value cannot be expressed as a **long** type integer, correct operation is not guaranteed.

ldiv_t ldiv (long numer, long denom) Quotient and Remainder

Description: Carries out division of **long** type integers and obtains the quotient and remainder.

Header file: <stdlib.h>

Return values: Quotient and remainder of division of **numer** by **denom**

Parameters:
numer Dividend
denom Divisor

Example:
```
#include <stdlib.h>
long numer, denom;
ldiv_t ret;
   ret=ldiv(numer, denom);
```

long long llabs (long long j) Absolute Value

Description: Calculates the absolute value of a **long long** type integer.

Header file: <stdlib.h>

Return values: Absolute value of j

Parameters: j Integer to calculate the absolute value of

Example:
```
#include <stdlib.h>
long long j;
long long ret;
    ret=llabs(j);
```

Remarks: If the resultant absolute value cannot be expressed as a **long long** type integer, correct operation is not guaranteed.

lldiv_t lldiv (long long numer, long long denom) Quotient and Remainder

Description: Carries out division of **long long** type integers and obtains the quotient and remainder.

Header file: <stdlib.h>

Return values: Quotient and remainder of division of **numer** by **denom**

Parameters: numer Dividend
denom Divisor

Example:
```
#include <stdlib.h>
long long numer, denom;
lldiv_t ret;
    ret=lldiv(numer,denom);
```

(14) <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.

Type	Definition Name	Description
Function	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 11.3, Standard Library Error Messages.

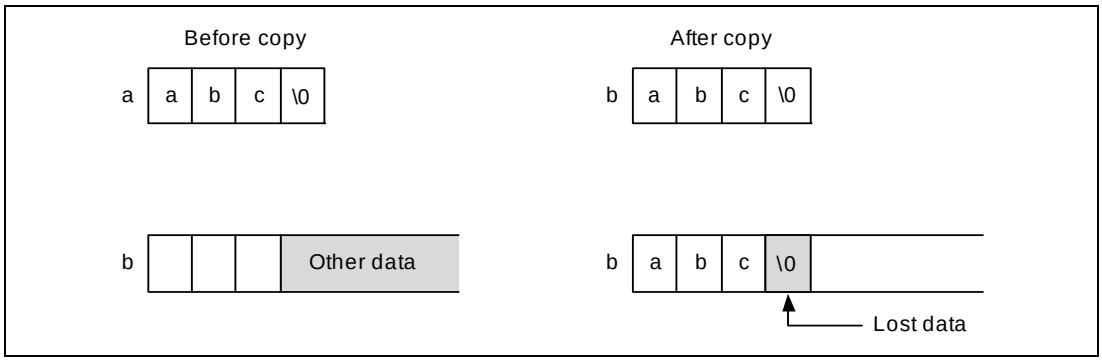
When using functions defined in this standard include file, note the following.

(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**.

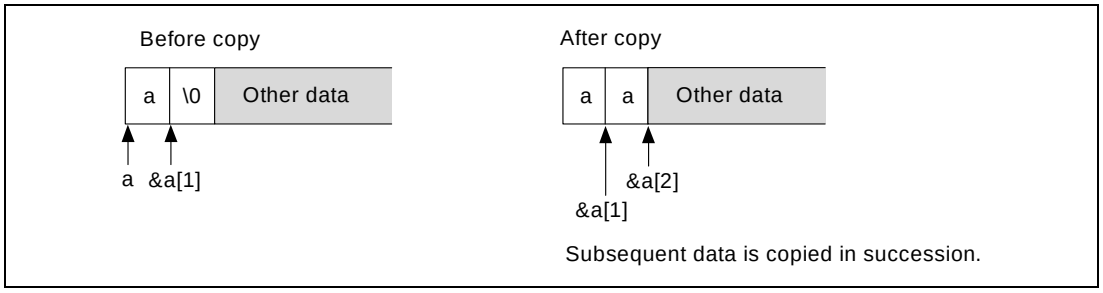

Before copy
After copy
a
a
b
c
\0
b
a
b
c
\0
b
Other data
b
a
b
c
\0
Lost data

(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.



void *memcpy (void *s1, const void *s2, size_t n) Storage Area Copy

Description: Copies the contents of a source storage area of a specified length to a destination storage area.

Header file: <string.h>

Return values: **s1** value

Parameters:

s1	Pointer to destination storage area
s2	Pointer to source storage area
n	Number of characters to be copied

Example:

```
#include <string.h>
void *ret, *s1;
const void *s2;
size_t n;
    ret=memcpy(s1, s2, n);
```

char *strcpy (char *s1, const char *s2) String Copy

Description: Copies the contents of a source string including the null character to a destination storage area.

Header file: <string.h>

Return values: **s1** value

Parameters:

s1	Pointer to destination storage area
s2	Pointer to source string

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
    ret=strcpy(s1, s2);
```

char *strncpy (char *s1, const char *s2, size_t n) String Copy

Description: Copies a source string of a specified length to a destination storage area.

Header file: <string.h>

Return values: **s1** value

Parameters:

s1	Pointer to destination storage area
s2	Pointer to source string
n	Number of characters to be copied

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
size_t n;
    ret=strncpy(s1, s2, n);
```

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.

char *strcat (char *s1, const char *s2) String Concatenation

Description: Concatenates a string after another string.

Header file: <string.h>

Return values: **s1** value

Parameters:

s1	Pointer to the string after which another string is appended
s2	Pointer to the string to be appended after the other string

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
    ret=strcat(s1, s2);
```

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.

char *strncat (char *s1, const char *s2, size_t n) String Concatenation

Description: Concatenates a string of a specified length after another string.

Header file: <string.h>

Return values: **s1** value

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

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
size_t n;
    ret=strncat(s1, s2, n);
```

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.

long memcmp (const void *s1, const void *s2, size_t n) Storage Area Comparison

Description: Compares the contents of two storage areas specified.

Header file: <string.h>

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

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

Example:

```
#include <string.h>
const void *s1, *s2;
size_t n;
int ret;
    ret=memcmp(s1, s2, n);
```

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.

long strcmp (const char *s1, const char *s2) String Comparison

Description: Compares the contents of two strings specified.

Header file: <string.h>

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

Example:

```
#include <string.h>
const char *s1, *s2;
int ret;
    ret=strcmp(s1, s2);
```

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.

long strncmp (const char *s1, const char *s2, size_t n) String Comparison

Description: Compares two strings specified up to a specified length.

Header file: <string.h>

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
n	Maximum number of characters to compare

Example:

```
#include <string.h>
const char *s1, *s2;
size_t n;
int ret;
    ret=strncmp(s1, s2, n);
```

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.

void *memchr (const void *s, long c, size_t n) Character Search in Storage Area

Description: Searches a specified storage area for the first occurrence of a specified character.

Header file: <string.h>

Return values: If the character is found: Pointer to the found character
If the character is not found: **NULL**

Parameters:

s	Pointer to the storage area to be searched
c	Character to search for
n	Number of characters to search

Example:

```
#include <string.h>
const void *s;
int c;
size_t n;
void *ret;
    ret=memchr(s, c, n);
```

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.

char *strchr (const char *s, long c) First Occurrence of Character

Description: Searches a specified string for the first occurrence of a specified character.

Header file: <string.h>

Return values: If the character is found: Pointer to the found character
If the character is not found: **NULL**

Parameters:

s	Pointer to the string to be searched
c	Character to search for

Example:

```
#include <string.h>
const char *s;
int c;
char *ret;
    ret=strchr(s, c);
```

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.

size_t strcspn (const char *s1, const char *s2) Number of Characters before First Occurrence of Specified Characters

Description: 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.

Header file: <string.h>

Return values: Number of characters at the beginning of the **s1** string that are not included in the **s2** string

Parameters:

s1	Pointer to the string to be checked
s2	Pointer to the string used to check **s1**

Example:

```
#include <string.h>
const char *s1, *s2;
size_t ret;
    ret=strcspn(s1, s2);
```

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.

char *strpbrk (const char *s1, const char *s2) First Occurrence of Specified Characters

Description: Searches a specified string for the first occurrence of the character that is included in another string specified.

Header file: <string.h>

Return values: If the character is found: Pointer to the found character
If the character is not found: **NULL**

Parameters:

s1	Pointer to the string to be searched
s2	Pointer to the string that indicates the characters to search **s1** for

Example:

```
#include <string.h>
const char *s1, *s2;
char *ret;
    ret=strpbrk(s1, s2);
```

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.

char *strrchr (const char *s, long c) Last Occurrence of Specified Character

Description: Searches a specified string for the last occurrence of a specified character.

Header file: <string.h>

Return values: If the character is found: Pointer to the found character
If the character is not found: **NULL**

Parameters:

s	Pointer to the string to be searched
c	Character to search for

Example:

```
#include <string.h>
const char *s;
int c;
char *ret;
    ret=strrchr(s, c);
```

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.

size_t strspn (const char *s1, const char *s2) Number of Consecutive Characters Specified

Description: Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are included in another string specified.

Header file: <string.h>

Return values: Number of characters at the beginning of the **s1** string that are included in the **s2** string

Parameters:

s1	Pointer to the string to be checked
s2	Pointer to the string used to check **s1**

Example:

```
#include <string.h>
const char *s1, *s2;
size_t ret;
    ret=strspn(s1, s2);
```

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.

char *strstr (const char *s1, const char *s2) First Occurrence of String

Description: Searches a specified string for the first occurrence of another string specified.

Header file: <string.h>

Return values: If the string is found: Pointer to the found string
If the string is not found: **NULL**

Parameters:

s1	Pointer to the string to be searched
s2	Pointer to the string to search for

Example:

```
#include <string.h>
const char *s1, *s2;
char *ret;
    ret=strstr(s1, s2);
```

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.

char *strtok (char *s1, const char *s2) Division into Tokens

Description: Divides a specified string into some tokens.

Header file: <string.h>

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

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
    ret=strtok(s1, s2);
```

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.'

void *memset (void *s, long c, size_t n) Character Repeating

Description: Sets a specified character a specified number of times at the beginning of a specified storage area.

Header file: <string.h>

Return values: Value of **s**

Parameters:

s	Pointer to storage area to set characters in
c	Character to be set
n	Number of characters to be set

Example:

```
#include <string.h>
void *s, *ret;
int c;
size_t n;
    ret=memset(s, c, n);
```

Remarks: The **memset** function sets the character specified by **c** a number of times specified by **n** in the storage area specified by **s**.

char *strerror (long s) Error Message String

Description: Returns an error message corresponding to a specified error number.

Header file: <string.h>

Return values: Pointer to the error message (string) corresponding to the specified error number

Parameters: s Error number

Example:
```
#include <string.h>
char *ret;
int s;
    ret=strerror(s);
```

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.

size_t strlen (const char *s) String Length

Description: Calculates the length of a string.

Header file: <string.h>

Return values: Number of characters in the string

Parameters: s Pointer to the string to check the length of

Example:
```
#include <string.h>
const char *s;
size_t ret;
    ret=strlen(s);
```

Remarks: The null character at the end of the **s** string is excluded from the string length.

void *memmove (void *s1, const void *s2, size_t n) Storage Area Move

Description: 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.

Header file: <string.h>

Return values: Value of **s1**

Parameters:

s1	Pointer to the destination storage area
s2	Pointer to the source storage area
n	Number of characters to be copied

Example:

```
#include <string.h>
void *ret, *s1
const void *s2;
size_t n;
     ret=memmove(s1, s2, n);
```

(15) <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	cacos	Calculates the arc cosine of a complex number.
	casin	Calculates the arc sine of a complex number.
	catan	Calculates the arc tangent of a complex number.
	ccos	Calculates the cosine of a complex number.
	csin	Calculates the sine of a complex number.
	ctan	Calculates the tangent of a complex number.
	cacosh	Calculates the arc hyperbolic cosine of a complex number.
	casinh	Calculates the arc hyperbolic sine of a complex number.
	catanh	Calculates the arc hyperbolic tangent of a complex number.
	ccosh	Calculates the hyperbolic cosine of a complex number.
	csinh	Calculates the hyperbolic sine of a complex number.
	ctanh	Calculates the hyperbolic tangent of a complex number.
	cexp	Calculates the natural logarithm base **e** raised to the complex power 2.
	clog	Calculates the natural logarithm of a complex number.
	cabs	Calculates the absolute value of a complex number.
	cpow	Calculates a power of a complex number.
	csqrt	Calculates the square root of a complex number.
	carg	Calculates the argument of a complex number.
	cimag	Calculates the imaginary part of a complex number.
	conj	Reverses the sign of the imaginary part and calculates the complex conjugate of a complex number.
	cproj	Calculates the projection of a complex number on Riemann sphere.
	creal	Calculates the real part of a complex number.

float complex cacosf(float complex z)
double complex cacos(double complex z)
long double complex cacosl(long double complex z) Complex Arc Cosine

Description: Calculates the arc cosine of a complex number.

Header file: <complex.h>

Return values: Normal: Complex arc cosine of **z**
Abnormal: Domain error: Returns not-a-number.

Parameters: z Complex number for which arc cosine is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=cacos(z);
```

Error conditions: A domain error occurs for a value of **z** not in the range [–1.0, 1.0].

Remarks: The **cacos** function returns the arc cosine in the range [0, π] on the real axis and in the infinite range on the imaginary axis.

float complex casinf(float complex z)
double complex casin(double complex z)
long double complex casinl(long double complex z) Complex Arc Sine

Description: Calculates the arc sine of a complex number.

Header file: <complex.h>

Return values: Normal: Complex arc sine of **z**
Abnormal: Domain error: Returns not-a-number.

Parameters: z Complex number for which arc sine is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=casin(z);
```

Error conditions: A domain error occurs for a value of **z** not in the range [–1.0, 1.0].

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.

float complex catanf(float complex z)
double complex catan(double complex z)
long double complex catanl(long double complex z) Complex Arc Tangent

Description: Calculates the arc tangent of a complex number.

Header file: <complex.h>

Return values: Normal: Complex arc tangent of **z**

Parameters: z Complex number for which arc tangent is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=catan(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.

float complex ccosf(float complex z)
double complex ccos(double complex z)
long double complex ccosl(long double complex z) Complex Cosine

Description: Calculates the cosine of a complex number.

Header file: <complex.h>

Return values: Complex cosine of **z**

Parameters: z Complex number for which cosine is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=ccos(z);
```

float complex csinf(float complex z)
double complex csin(double complex z)
long double complex csinl(long double complex z) Complex Sine

Description: Calculates the sine of a complex number.

Header file: <complex.h>

Return values: Complex sine of **z**

Parameters: z Complex number for which sine is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=csin(z);
```

float complex ctanf(float complex z)
double complex ctan(double complex z)
long double complex ctanl(long double complex z) Complex Tangent

Description: Calculates the tangent of a complex number.

Header file: <complex.h>

Return values: Complex tangent of **z**

Parameters: z Complex number for which tangent is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=ctan(z);
```

float complex cacoshf(float complex z)
double complex cacosh(double complex z)
long double complex cacoshl(long double complex z) Complex Arc Hyperbolic Cosine

Description: Calculates the arc hyperbolic cosine of a complex number.

Header file: <complex.h>

Return values:
Normal:	Complex arc hyperbolic cosine of **z**
Abnormal:	Domain error: Returns not-a-number.

Parameters:
z	Complex number for which arc hyperbolic cosine is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=cacosh(z);
```

Error conditions: A domain error occurs for a value of **z** not in the range [–1.0, 1.0].

Remarks: The **cacoshf** function returns the arc hyperbolic cosine in the range [0, π].

float complex casinhf(float complex z)
double complex casinh(double complex z)
long double complex casinhl(long double complex z) Complex Arc Hyperbolic Sine

Description: Calculates the arc hyperbolic sine of a complex number.

Header file: <complex.h>

Return values: Complex arc hyperbolic sine of **z**

Parameters:
z	Complex number for which arc hyperbolic sine is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=casinh(z);
```

float complex catanhf(float complex z)
double complex catanh(double complex z)
long double complex catanhl(long double complex z) Complex Arc Hyperbolic Tangent

Description: Calculates the arc hyperbolic tangent of a complex number.

Header file: <complex.h>

Return values: Complex arc hyperbolic tangent of **z**

Parameters: z Complex number for which arc hyperbolic tangent is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=catanh(z);
```

float complex ccoshf(float complex z)
double complex ccosh(double complex z)
long double complex ccoshl(long double complex z) Complex Hyperbolic Cosine

Description: Calculates the hyperbolic cosine of a complex number.

Header file: <complex.h>

Return values: Complex hyperbolic cosine of **z**

Parameters: z Complex number for which hyperbolic cosine is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=ccosh(z);
```

float complex csinhf(float complex z)
double complex csinh(double complex z)
long double complex csinhl(long double complex z) Complex Hyperbolic Sine

Description: Calculates the hyperbolic sine of a complex number.

Header file: <complex.h>

Return values: Complex hyperbolic sine of **z**

Parameters: z Complex number for which hyperbolic sine is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=csinh(z);
```

float complex ctanhf(float complex z)
double complex ctanh(double complex z)
long double complex ctanhl(long double complex z) Complex Hyperbolic Tangent

Description: Calculates the hyperbolic tangent of a complex number.

Header file: <complex.h>

Return values: Complex hyperbolic tangent of **z**

Parameters: z Complex number for which hyperbolic tangent is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=ctanh(z);
```

float complex cexpf(float complex z)
double complex cexp(double complex z)
long double complex cexpl(long double complex z) Complex Exponential Function

Description: Calculates the exponential function value of a complex number.

Header file: <complex.h>

Return values: Exponential function value of **z**

Parameters:

z		Complex number for which exponential function is to be computed

Example:

```
#include <complex.h>
double complex z, ret;
    ret=cexp(z);
```

float complex clogf(float complex z)
double complex clog(double complex z)
long double complex clogl(long double complex z) Complex Natural Logarithm

Description: Calculates the natural logarithm of a complex number.

Header file: <complex.h>

Return values:

Normal:	Natural logarithm of **z**
Abnormal:	Domain error: Returns not-a-number.

Parameters:

z		Complex number for which natural logarithm is to be computed

Example:

```
#include <complex.h>
double complex z, ret;
    ret=clog(z);
```

Error conditions: A domain error occurs if **z** is negative.
A range error occurs if **z** is 0.0.

Remarks: 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.

float cabsf(float complex z)
double cabs(double complex z)
long double cabsl(long double complex z) Complex Absolute Value

Description: Calculates the absolute value of a complex number.

Header file: <complex.h>

Return values: Absolute value of **z**

Parameters:

z	Complex number for which absolute value is to be computed	

Example:

```
#include <complex.h>
double complex z, ret;
    ret=cabs(z);
```

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) Complex Power

Description: Calculates a power of a complex number.

Header file: <complex.h>

Return values:

Normal:	Value of **x** raised to the power **y**
Abnormal:	Domain error: Returns not-a-number.

Parameters:

x	Value to be raised to a power
y	Power value

Example:

```
#include <complex.h>
double complex x, y;
    ret=cpow(x, y);
```

Error conditions: 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.

Remarks: The branch cut for the first parameter of the **cpow** function group is along the negative real axis.

float complex csqrtf(float complex z)
double complex csqrt(double complex z)
long double complex csqrtl(long double complex z) Complex Square Root

Description: Calculates the square root of a complex number.

Header file: <complex.h>

Return values: Normal: Complex square root of **z**
Abnormal: Domain error: Returns not-a-number.

Parameters: z Complex number for which the square root is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=csqrt(z);
```

Error conditions: A domain error occurs if **z** is negative.

Remarks: 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.

float cargf(float complex z)
double carg(double complex z)
long double cargl(long double complex z) Argument

Description: Calculates the argument.

Header file: <complex.h>

Return values: Argument value of **z**

Parameters: z Complex number for which the argument is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=carg(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 [–π, +π].

float cimagf(float complex z)
double cimag(double complex z)
long double cimagl(long double complex z) Imaginary Part

Description: Calculates the imaginary part.

Header file: <complex.h>

Return values: Imaginary part value of **z** as a real number

Parameters: z Complex number for which the imaginary part is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=cimag(z);
```

float complex conjf(float complex z)
double complex conj(double complex z)
long double complex conjl(long double complex z) Complex Conjugate

Description: Reverses the sign of the imaginary part of a complex number and calculates the complex conjugate.

Header file: <complex.h>

Return values: Complex conjugate of **z**

Parameters: z Complex number for which the complex conjugate is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=conj(z);
```

float complex cprojf(float complex z)
double complex cproj(double complex z)
long double complex cprojl(long double complex z) Projection on Riemann Sphere

Description: Calculates the projection of a complex number on the Riemann sphere.

Header file: <complex.h>

Return values: Projection of **z** on the Riemann sphere

Parameters: z Complex number for which the projection on the Riemann sphere is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=cproj(z);
```

float crealf(float complex z)
double creal(double complex z)
long double creall(long double complex z) Real Part

Description: Calculates the real part of a complex number.

Header file: <complex.h>

Return values: Real part value of **z**

Parameters: z Complex number for which the real part value is to be computed

Example:
```
#include <complex.h>
double complex z, ret;
    ret=creal(z);
```

(16) <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.

long feclearexcept(long e) Clearing Exception

Description: Attempts to clear a floating-point exception.

Header file: <fenv.h>

Return values: Normal: 0
Abnormal: Nonzero

Parameters: e Floating-point exception

Example:
```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
int ret, e;
    ret=feclearexcept(e);
```

Remarks: Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

long fegetexceptflag(fexcept_t *f, long e) Getting Exception Flag State

Description: Gets the state of a floating-point flag.

Header file: <fenv.h>

Return values: Normal: 0
Abnormal: Nonzero

Parameters: f Pointer to area to store the exception flag state
e Value indicating the exception flag whose state is to be acquired

Example:
```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
int ret;
fexcept_t f;
    ret=fegetexceptflag(&f, e);
```

Remarks: Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

long feraiseexcept(long e) Generating Exception

Description: Attempts to generate a floating-point exception.

Header file: <fenv.h>

Return values: Normal: 0
Abnormal: Nonzero

Parameters: e Value indicating the exception to be generated

Example:
```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
int ret, e;
    ret=feraiseexcept(e);
```

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.

long fesetexceptflag(const fexcept_t *f, long e) Setting Exception Flag State

Description: Sets the state of an exception flag.

Header file: <fenv.h>

Return values:
Normal:	0
Abnormal:	Nonzero

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

Example:
```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
fexcept_t f;
fegetexceptflag(&f, FE_OVERFLOW) /* Saves flag state */
fesetexceptflag(&f,FE_OVERFLOW); /* Sets flag state */
```

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.

long fetestexcept(long e) Checking Exception Flag States

Description: Checks the exception flag states.

Header file: <fenv.h>

Return values: Bitwise OR of **e** and floating-point exception macros

Parameters:

e	Value indicating flags whose states are to be checked (multiple flags can be specified)

Example:

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
int e = fetestexcept(FE_INVALID | FE_OVERFLOW);
if (e & FE_INVALID) fnc1();
if (e & FE_OVERFLOW) fnc2();
```

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.

long fegetround(void) Getting Rounding Direction

Description: Gets the current rounding direction.

Header file: <fenv.h>

Return values:

Normal:	0
Abnormal:	Negative value when there is no rounding direction macro value or the rounding direction cannot be determined

Example:

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
int ret = fgetround();
```

Remarks: Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

long fesetround(long rnd) Setting Rounding Direction

Description: Sets the current rounding direction.

Header file: <fenv.h>

Return values: 0 only when the rounding direction has been set successfully

Example:

```
#include <fenv.h>
#include <assert.h>
void f(int round_dir)
{
#pragma STDC FENV_ACCESS ON
    int save_round;
    int setround_ok;
    save_round = fegetround();
    setround_ok = fesetround(round_dir);
    assert(setround_ok == 0);
    fesetround(save_round);
}
```

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.

long fegetenv(fenv_t *f) Getting Floating-Point Environment

Description: Gets the floating-point environment.

Header file: <fenv.h>

Return values: Normal: 0
Abnormal: Nonzero

Parameters: f Pointer to area to store the floating-point environment

Example:
```
#include <fenv.h>
int ret, fenv_t f;
    ret=fegetenv(f);
```

Remarks: Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

long feholdexcept(fenv_t *f) Saving Floating-Point Environment

Description: Saves the floating-point environment.

Header file: <fenv.h>

Return values: 0 only when the environment has been saved successfully

Parameters: f Pointer to the floating-point environment

Example:
```
#include <fenv.h>
int ret, fenv_t f;
    ret=feholdexcept(&f);
```

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.

Remarks: Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

long fesetenv(const fenv_t *f) Setting Floating-Point Environment

Description: Sets the floating-point environment.

Header file: <fenv.h>

Return values: Normal: 0
Abnormal: Nonzero

Parameters: f Pointer to the floating-point environment

Example:
```
#include <fenv.h>
int ret, fenv_t f;
    ret=fesetenv(f);
```

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.

long feupdateenv(const fenv_t *f) Setting Floating-Point Environment

Description: Sets the floating-point environment with the previously generated exceptions retained.

Header file: <fenv.h>

Return values: Normal: 0
Abnormal: Nonzero

Parameters: f Pointer to the floating-point environment to be set

Example:
```
#include <fenv.h>
double f(double x)
{
#pragma STDC FENV_ACCESS ON
    double ret;
    fenv_t prev_env;
    if (feholdexcept(&prev_env))
        return /* The environment has a problem */;
    // Calculates ret
    if (/* Checks if it is a pseudo underflow */)
        if (feclearexcept(FE_UNDERFLOW))
            return /* The environment has a problem */;
        if (feupdateenv(&prev_env))
            return /* The environment has a problem */;
    return ret;
}
```

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.

(17) <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.
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	

Type	Definition Name	Description
Variable (macro)	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.

intmax_t imaxabs(intmax_t a) Absolute Value

Description: Calculates the absolute value.

Header file: <inttypes.h>

Return values: Absolute value of **a**

Parameters: a Value for which the absolute value is to be computed

Example:
```
#include <inttypes.h>
intmax a, ret;
    ret=imaxabs(a);
```

intmaxdiv_t imaxdiv(intmax_t n, intmax_t d) Division

Description: Performs a division operation.

Header file: <inttypes.h>

Return values: Division result consisting of the quotient and remainder

Parameters: n Dividend and divisor
d

Example:
```
#include <inttypes.h>
intmax_t n, m;
```

```
intmaxdiv_t ret;
    ret=imaxdiv(n, m);
```

intmax_t strtoimax(const char *nptr, char **endptr, long base) Converting String
uintmax_t strtoumax(const char *nptr, char **endptr, long base) to intmax_t Type

Description: Converts a number-representing string to an **intmax_t** type integer.

Header file: <inttypes.h>

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**

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)

Example:

```
#include <inttypes.h>
intmax_t ret;
const char *nptr;
char **endptr;
int base;
    ret=strtoimax(nptr,endptr,base);
```

Error conditions: If the converted result overflows, **ERANGE** is set in **errno**.

Remarks: 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.

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) Converting Wide String to Integer

Description: Converts a number-representing string to an **intmax_t** or **uintmax_t** type integer.

Header file: <stddef.h>, <inttypes.h>

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**

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)

Example:

```
#include <stddef.h>
#include <inttypes.h>
intmax_t ret;
const char *nptr;
char **endptr;
int base;
    ret=wcstoimax(nptr,endptr,base);
```

Error conditions: If the converted result overflows, **ERANGE** is set in **errno**.

Remarks: 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.

(18) <iso646.h>

This header file defines macros only.

Type	Definition Name	Description
Macro	and	&&
	and_eq	&=
	bitand	&
	bitor	|
	compl	~
	not	!
	not_eq	!=
	or	||
	or_eq	|=
	xor	^
	xor_eq	^=

(19) <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.

(20) <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	Indicates the minimum value of exact-width signed integer type.
	INT*N*_MAX	Indicates the maximum value of exact-width signed integer type.
	UINT*N*_MAX	Indicates the maximum value of exact-width unsigned integer type.

Type	Definition Name	Description
Macro	INT_LEAST*N*_MIN	Indicates the minimum value of minimum-width signed integer type.
	INT_LEAST*N*_MAX	Indicates the maximum value of minimum-width signed integer type.
	UINT_LEAST*N*_MAX	Indicates the maximum value of minimum-width unsigned integer type.
	INT_FAST*N*_MIN	Indicates the minimum value of fastest minimum-width signed integer type.
	INT_FAST*N*_MAX	Indicates the maximum value of fastest minimum-width signed integer type.
	UINT_FAST*N*_MAX	Indicates the maximum value of fastest minimum-width unsigned integer type.
	INTPTR_MIN	Indicates the minimum value of pointer-holding signed integer type.
	INTPTR_MAX	Indicates the maximum value of pointer-holding signed integer type.
	UINTPTR_MAX	Indicates the maximum value of pointer-holding unsigned integer type.
	INTMAX_MIN	Indicates the minimum value of greatest-width signed integer type.
	INTMAX_MAX	Indicates the maximum value of greatest-width signed integer type.
	UINTMAX_MAX	Indicates the maximum value of greatest-width unsigned integer type.
	PTRDIFF_MIN	–65535
	PTRDIFF_MAX	+65535
	SIG_ATOMIC_MIN	–127
	SIG_ATOMIC_MAX	+127
	SIZE_MAX	65535
	WCHAR_MIN	0
	WCHAR_MAX	65535U
	WINT_MIN	0
	WINT_MAX	4294967295U

Type	Definition Name	Description
Function (macro)	INT*N*_C	Expanded to an integer constant expression corresponding to **Int_least*N*_t**.
	UINT*N*_C	Expanded to an integer constant expression corresponding to **Uint_least*N*_t**.
	INT_MAX_C	Expanded to an integer constant expression with type **intmax_t**.
	UINT_MAX_C	Expanded to an integer constant expression with type **uintmax_t**.

(21) <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	—

Type-Generic Macro	<math.h> Functions	<complex.h> Functions
fmax	fmax	—
fmin	fmin	—
fmod	fmod	—
frexp	frexp	—
hypot	hypot	—
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

Type-Generic Macro	<math.h> Functions	<complex.h> Functions
creal	—	creal

(22) <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	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	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	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.

Type	Definition Name	Description
Function	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	These convert the initial part of a wide string to **long int**, **long long int**, **unsigned long int**, or **unsigned long long int** representation.
	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.

Type	Definition Name	Description
Function	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.
	wctob	Checks if a multibyte character representation can be converted to 1-byte representation.
	mbsinit	Checks if a specified object indicates the initial conversion state.
	mbrlen	Calculates the number of bytes in a multibyte character.
	mbrtowc	Converts a multibyte character to a wide character.
	wcrtomb	Converts a wide character to a multibyte character.
	mbsrtowcs	Converts a sequence of multibyte characters to a sequence of corresponding wide characters.
	wcsrtombs	Converts a sequence of wide characters to a sequence of corresponding multibyte characters.

long fwprintf(FILE *restrict fp,
const wchar_t *restrict control [, arg] …) Formatted Wide Character Output to File

Description: Outputs data to a stream input/output file according to the format.

Header file: <stdio.h>, <wchar.h>

Return values:

Normal:	Number of wide strings converted and output
Abnormal:	Negative value

Parameters:

fp	File pointer
control	Pointer to wide string indicating format
arg, …	List of data to be output according to format

Example:

```
#include <stdio.h>
#include <wchar.h>
FILE *fp;
const wchar_t *control=L”%ls”;
int ret;
wchar_t buffer[]=L”Hello World\n”;
    ret=fwprintf(fp, control, buffer);
```

Remarks: The **fwprintf** function is the wide-character version of the **fprintf** function.

long vfwprintf(FILE *restrict fp, const char *restrict control, va_list arg) Variable-Parameter Formatted Wide Character Output to File

Description: Outputs a variable parameter list to the specified stream input/output file according to a format.

Header file: <stdarg.h>, <stdio.h>, <wchar.h>

Return values:

Normal:	Number of characters converted and output
Abnormal:	Negative value

Parameters:

fp	File pointer
control	Pointer to wide string indicating format
arg	Parameter list

Example:

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
FILE *fp;
const wchar_t *control=L"%d";
int  ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vfprintf(fp, control, ap);
   va_end(ap);
}
```

Remarks: The **vfwprintf** function is the wide-character version of the **vfprintf** function.

long swprintf(wchar_t *restrict s, size_t n,
const wchar_t *restrict control [, arg] …) Formatted Wide String Output

Description: Converts data according to a format and outputs it to the specified area.

Header file: <stdio.h>, <wchar.h>

Return values:

Normal:	Number of characters converted
Abnormal:	When a representation format error occurs or writing **n** or more wide characters is requested: Negative value

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

Example:

```
#include <stdio.h>
#include <wchar.h>
wchar_t s*;
size_t n=12;
const wchar_t *control=”%ls”;
int ret;
wchar_t buffer[]=”Hello World\n”;
    ret=swprintf(s, n, control, buffer);
```

Error conditions: A representation format error occurs if an illegal multibyte string is passed to the **mbrtowc()** function.

Remarks: The **swprintf** function is the wide-character version of the **sprintf** function.

long vswprintf(wchar_t *restrict s, size_t n,
const wchar_t *restrict control, va_list arg) Variable Parameter Wide String Output

Description: Outputs a variable parameter list to the specified storage area according to a format.

Header file: <stdarg.h>, <wchar.h>

Return values:

Normal:	Number of characters converted
Abnormal:	Negative value

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

Example:

```
#include <stdarg.h>
#include <wchar.h>
wchar_t *s;
const wchar_t *control=L"%d";
int ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++) {
      ret=vswprintf(s, control, ap);
        va_arg(ap,int);
        s += ret;
     }
}
```

Remarks: The **vswprintf** function is the wide-character version of the **vsprintf** function.

long wprintf(const wchar_t *restrict control [, arg] …) Formatted Wide Character Output

Description: Converts data according to a format and outputs it to the standard output file (**stdout**).

Header file: <stdio.h>, <wchar.h>

Return values:
Normal:	Number of wide characters converted and output
Abnormal:	Negative value

Parameters:
control	Pointer to string indicating format
arg,…	Data to be output according to format

Example:
```
#include <stdio.h>
#include <wchar.h>
const wchar_t *control=L"%ls";
int ret;
wchar_t buffer[]=L"Hello World\n";
    ret=wprintf(control,buffer);
```

Remarks: The **wprintf** function is the wide-character version of **printf** function.

**long vwprintf(const wchar_t *restrict control,
va_list arg)** Variable-Parameter Wide Character Output

Description: Outputs a variable parameter list to the standard output file (**stdout**) according to a format.

Header file: <stdarg.h>, <wchar.h>

Return values:

Normal:	Number of characters converted and output
Abnormal:	Negative value

Parameters:

control	Pointer to wide string indicating format
arg	Parameter list

Example:

```
#include <stdarg.h>
#include <wchar.h>
FILE *fp;
const wchar_t *control=L"%d";
int ret;

void wprlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vwprintf(control, ap);
   va_end(ap);
}
```

Remarks: The **vwprintf** function is the wide-character version of the **vprintf** function.

long fwscanf(FILE *restrict fp, const wchar_t *restrict control [, ptr] …) Formatted Wide Character Input from File

Description: Inputs data from a stream input/output file and converts it according to a format.

Header file: <stdio.h>, <wchar.h>

Return values:

Normal:	Number of data items successfully input and converted
Abnormal:	Input data ends before input data conversion is performed: **EOF**

Parameters:

fp	File pointer
control	Pointer to wide string indicating format
ptr	Pointer to storage area that stores input data

Example:

```
#include <stdio.h>
#include <wchar.h>
FILE *fp;
const wchar_t *control=L”%d”;
int ret, buffer[10];
    ret=fwscanf(fp, control, buffer);
```

Remarks: The **fwscanf** function is the wide-character version of the **fscanf** function.

long vfwscanf(FILE *restrict fp, const wchar_t *restrict control, va_list arg) Variable-Parameter Formatted Wide Character Input from File

Description: Inputs data from a stream input/output file and converts it according to a format.

Header file: <stdarg.h>, <stdio.h>, <wchar.h>

Return values:

Normal:	Number of data items successfully input and converted
Abnormal:	Input data ends before input data conversion is performed: **EOF**

Parameters:

fp	File pointer
control	Pointer to wide string indicating format
arg	Parameter list

Example:

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
FILE *fp;
const wchar_t *control=L"%d";
int ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vfwscanf(fp, control, ap);
   va_end(ap);
}
```

Remarks: The **vfwscanf** is the wide-character version of the **vfscanf** function.

long swscanf(const wchar_t *restrict s, const wchar_t *restrict control [, ptr] …) Formatted Wide String Input

Description: Inputs data from the specified storage area and converts it according to a format.

Header file: <stdio.h>, <wchar.h>

Return values:

Normal:	Number of data items successfully input and converted
Abnormal:	**EOF**

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

Example:

```
#include <stdio.h>
#include <wchar.h>
const wchar_t *s, *control=L"%d";
int ret,buffer[10];
ret=swscanf(s, control, buffer);
```

Remarks: The **swscanf** is the wide-character version of the **sscanf** function.

long vswscanf(const wchar_t *restrict s, const wchar_t *restrict control, va_list arg) Variable-Parameter Formatted Wide String Input

Description: Inputs data from the specified storage area and converts it according to a format.

Header file: <stdarg.h>, <wchar.h>

Return values:
Normal:	Number of data items successfully input and converted
Abnormal:	**EOF**

Parameters:
s	Storage area containing data to be input
control	Pointer to wide string indicating format
arg	Parameter list

Example:
```
#include <stdarg.h>
#include <wchar.h>
const wchar_t *s, *control=L"%d";
int ret,buffer[10];
ret=vswscanf(s, control, buffer);
```

long wscanf(const wchar_t *control [, ptr] …) Formatted Wide Character Input

Description: Inputs data from the standard input file (**stdin**) and converts it according to a format.

Header file: <wchar.h>

Return values:
Normal:	Number of data items successfully input and converted
Abnormal:	**EOF**

Parameters:
control	Pointer to wide string indicating format
ptr,…	Pointer to storage area that stores input and converted data

Example:
```
#include <wchar.h>
const wchar_t *control=L"%d";
int ret,buffer[10];
    ret=wscanf(control, buffer);
```

Remarks: The **wscanf** function is the wide-character version of the **scanf** function.

long vwscanf(const wchar_t *restrict control,
va_list arg) Variable-Parameter Formatted Wide Character Input from File

Description: Inputs data from the specified storage area and converts it according to a format.

Header file: <stdarg.h>, <wchar.h>

Return values:

Normal:	Number of data items successfully input and converted
Abnormal:	Input data ends before input data conversion is performed: **EOF**

Parameters:

control	Pointer to wide string indicating format
arg	Parameter list

Example:

```
#include <stdarg.h>
#include <wchar.h>

FILE *fp;
const wchar_t *control=L"%d";
int ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vwscanf(control, ap);
   va_end(ap);
}
```

Remarks: The **vwscanf** function is provided to support wide-character format with the **vscanf** function.

wint_t fgetwc(FILE *fp) One Wide Character Input from File

Description: Inputs one wide character from a stream input/output file.

Header file: <stdio.h>, <wchar.h>

Return values:
Normal: End-of-file: **EOF**
Otherwise: Input wide character
Abnormal: **EOF**

Parameters:
fp File pointer

Example:
```
#include <stdio.h>
#include <wchar.h>
FILE *fp;
wint_t ret;
    ret=fgetwc(fp);
```

Error conditions: When a read error occurs, the error indicator for that file is set.

Remarks: The **fgetwc** function is provided to support wide-character input to the **fgetc** function.

wchar_t *fgetws(wchar_t *restrict s, long n, FILE *fp) Wide String Input from File

Description: Inputs a wide string from a stream input/output file.

Header file: <stdio.h>, <wchar.h>

Return values:

Normal:	End-of-file: **NULL**
	Otherwise: **s**
Abnormal:	**NULL**

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

Example:

```
#include <stdio.h>
#include <wchar.h>
wchar_t *s, *ret;
int n;
FILE *fp;
    ret=fgetws(s,n,fp);
```

Remarks: The **fgetws** function is provided to support wide-character input to the **fgets** function.

wint_t fputwc(wchar_t c, FILE *fp) One Wide Character Output to File

Description: Outputs one wide character to a stream input/output file.

Header file: <stdio.h>, <wchar.h>

Return values:
Normal:	Output wide character
Abnormal:	**EOF**

Parameters:
c	Character to be output
fp	File pointer

Example:
```
#include <stdio.h>
#include <wchar.h>
FILE *fp;
wchar_t c;
wint_t ret;
    ret=fputwc(c,fp);
```

Error conditions: When a write error occurs, the error indicator for that file is set.

Remarks: The **fputwc** function is the wide-character version of the **fputc** function.

long fputws(const wchar_t *restrict s, FILE *restrict fp) Wide String Output to File

Description: Outputs a wide string to a stream input/output file.

Header file: <stdio.h>, <wchar.h>

Return values:

Normal:	0
Abnormal	**EOF**

Parameters:

s	Pointer to wide string to be output
fp	File pointer

Example:

```
#include <stdio.h>
#include <wchar.h>
const wchar_t *s;
int ret;
FILE *fp;
    ret=fputws(s,fp);
```

Remarks: The **fputws** function is the wide-character version of the **fputs** function.

long fwide(FILE *fp, long mode) Specifying Input Unit of File

Description: Specifies the input unit of a file.

Header file: <stdio.h>, <wchar.h>

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

Parameters:

fp	File pointer
mode	Value indicating the input unit

Example:

```
#include <stdio.h>
#include <wchar.h>
FILE *fp;
int mode, ret;
    ret=fwide(fp,mode);
```

Remarks: The **fwide** function does not change the stream input/output unit that has already been determined.

long getwc(FILE *fp) One Wide Character Input from File

Description: Inputs one wide character from a stream input/output file.

Header file: <stdio.h>, <wchar.h>

Return values: Normal: End-of-file: **WEOF**
Otherwise: Input wide character
Abnormal: **EOF**

Parameters: fp File pointer

Example:
```
#include <stdio.h>
#include <wchar.h>
FILE *fp;
int ret;
    ret=getwc(fp);
```

Error conditions: When a read error occurs, the error indicator for that file is set.

Remarks: 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**.

long getwchar(void) One Wide Character Input

Description: Inputs one wide character from the standard input file (**stdin**).

Header file: <wchar.h>

Return values: Normal: End-of-file: **WEOF**
Otherwise: Input wide character
Abnormal: **EOF**

Example:
```
#include <wchar.h>
int ret;
    ret=getwchar();
```

Error conditions: When a read error occurs, the error indicator for that file is set.

Remarks: The **getwchar** function is the wide-character version of the **getchar** function.

wint_t putwc(wchar_t c, FILE *fp) One Wide Character Output to File

Description: Outputs one wide character to a stream input/output file.

Header file: <stdio.h>, <wchar.h>

Return values:

Normal:	Output wide character
Abnormal:	**WEOF**

Parameters:

c	Wide character to be output
fp	File pointer

Example:

```
#include <stdio.h>
#include <wchar.h>
FILE *fp;
wchar_t c;
wint_t ret;
    ret=putwc(c,fp);
```

Error conditions: When a write error occurs, the error indicator for that file is set.

Remarks: 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**.

wint_t putwchar(wchar_t c) One Wide Character Output

Description: Outputs one wide character to the standard output file (**stdout**).

Header file: <wchar.h>

Return values: Normal: Output wide character
Abnormal: **WEOF**

Parameters: c Wide character to be output

Example:
```
#include <wchar.h>
wint_t ret;
wchar_t c;
    ret=putwchar(c);
```

Error conditions: When a write error occurs, the error indicator for that file is set.

Remarks: The **putwchar** function is the wide-character version of the **putchar** function.

wint_t ungetwc(wint_t c, FILE *fp) One Wide Character Return to File

Description: Returns one wide character to a stream input/output file.

Header file: <stdio.h>, <wchar.h>

Return values: Normal: Returned wide character
Abnormal: **WEOF**

Parameters: c Wide character to be returned
fp File pointer

Example:
```
#include <stdio.h>
#include <wchar.h>
wint_t ret;
wchar_t c;
FILE *fp;
    ret=ungetwc(c,fp);
```

Remarks: The **ungetwc** function is the wide-character version of the **ungetc** function.

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) Wide String Conversion to Floating-Point Number

Description: Converts the initial part of a wide string to a specified-type floating-point number.

Header file: <wchar.h>

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 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

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

Example:

```
#include <wchar.h>
const wchar_t *nptr;
wchar_t **endptr;
double ret;
    ret=wcstod(nptr,endptr);
```

Error conditions: If the converted result overflows or underflows, **errno** is set.

Remarks: The **wcstod** function group is the wide-character version of the **strtod** function group.

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) Wide String Conversion to Integer

Description: Converts the initial part of a wide string to a specified-type integer.

Header file: <wchar.h>

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 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

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)

Example:

```
#include <wchar.h>
long ret;
const wchar_t *nptr;
wchar_t **endptr;
int base;
    ret=wcstoull(nptr,endptr,base);
```

Error conditions: If the converted result overflows, **errno** is set.

Remarks: The **wcstol** function group is the wide-character version of the **strtol** function group.

wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2) Wide String Copy

Description: Copies the contents of a source wide string including the null character to a destination storage area.

Header file: <wchar.h>

Return values: **s1** value

Parameters:

s1	Pointer to destination storage area
s2	Pointer to source string

Example:

```
#include <wchar.h>
wchar_t *s1, *ret;
const wchar_t *s2;
    ret=wcscpy(s1,s2);
```

Remarks: The **wcscpy** function group is the wide-character version of the **strcpy** function group.

wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n) Wide String Copy

Description: Copies a source wide string of a specified length to a destination storage area.

Header file: <wchar.h>

Return values: **s1** value

Parameters:

s1	Pointer to destination storage area
s2	Pointer to source string
n	Number of characters to be copied

Example:

```
#include <wchar.h>
wchar_t *s1, *ret;
const wchar_t *s2;
size_t n;
    ret=wcsncpy(s1,s2,n);
```

Remarks: The **wcsncpy** function is the wide-character version of the **strncpy** function.

wchar_t *wmemcpy(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n) Storage Area Copy

Description: Copies the contents of a source storage area of a specified length to a destination storage area.

Header file: <wchar.h>

Return values: **s1** value

Parameters:

s1	Pointer to destination storage area
s2	Pointer to source storage area
n	Number of characters to be copied

Example:

```
#include <wchar.h>
wchar_t *ret, *s1;
const wchar_t *s2;
size_t n;
    ret=wmemcpy(s1,s2,n);
```

Remarks: The **wmemcpy** function is the wide-character version of the **memcpy** function.

wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n) Storage Area Move

Description: 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.

Header file: <wchar.h>

Return values: **s1** value

Parameters:

s1	Pointer to destination storage area
s2	Pointer to source storage area
n	Number of characters to be copied

Example:

```
#include <wchar.h>
wchar_t *ret, *s1;
const wchar_t *s2;
size_t n;
    ret=wmemmove(s1,s2,n);
```

Remarks: The **wmemmove** function is the wide-character version of the **memmove** function.

wchar_t *wcscat(wchar_t *s1, const wchar_t *s2) Wide String Concatenation

Description: Concatenates a string after another string.

Header file: <wchar.h>

Return values: **s1** value

Parameters:

s1	Pointer to the string after which another string is appended
s2	Pointer to the string to be appended after the other string

Example:

```
#include <wchar.h>
wchar_t *s1, *ret;
const wchar_t *s2;
    ret=wcscat(s1,s2);
```

Remarks: The **wcscat** function is the wide-character version of the **strcat** function.

wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n) Wide String Concatenation

Description: Concatenates a string of a specified length after another string.

Header file: <wchar.h>

Return values: **s1** value

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

Example:

```
#include <wchar.h>
wchar_t *s1, *ret;
const wchar_t *s2;
size_t n;
    ret=wcsncat(s1,s2,n);
```

Remarks: The **wcsnca**t function is the wide-character version of the **strncat** function.

long wcscmp(const wchar_t *s1, const wchar_t *s2) String Comparison

Description: Compares the contents of two strings specified.

Header file: <wchar.h>

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

Example:

```
#include <wchar.h>
const wchar_t *s1, *s2;
```

```
int ret;
    ret=wcscmp(s1,s2);
```

Remarks: The **wcscmp** function is the wide-character version of the **strcmp** function.

long wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n) String Comparison

Description: Compares two strings specified up to a specified length.

Header file: <wchar.h>

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
n	Maximum number of characters to compare

Example:

```
#include <wchar.h>
const wchar_t *s1, *s2;
size_t n;
int ret;
    ret=wcsncmp(s1,s2,n);
```

Remarks: The **wcsncmp** function is the wide-character version of the **strncmp** function.

long wmemcmp(const wchar_t * s1, const wchar_t * s2, size_t n) Storage Area Comparison

Description: Compares the contents of two storage areas specified.

Header file: <wchar.h>

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

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

Example:

```
#include <wchar.h>
const wchar_t *s1, *s2;
size_t n;
int ret;
    ret=wmemcmp(s1,s2,n);
```

Remarks: The **wmemcmp** function is the wide-character version of the **memcmp** function.

wchar_t *wcschr(const wchar_t *s, wchar_t c) First Occurrence of Character

Description: Searches a specified string for the first occurrence of a specified character.

Header file: <wchar.h>

Return values: If the character is found: Pointer to the found character
If the character is not found: **NULL**

Parameters:

s	Pointer to the string to be searched
c	Character to search for

Example:

```
#include <wchar.h>
const wchar_t *s;
int c;
char *ret;
    ret=wcschr(s,c);
```

Remarks: The **wcschr** function is the wide-character version of the **strchr** function.

size_t wcscspn(const wchar_t *s1, const wchar_t *s2) Number of Characters before First Occurrence of Specified Characters

Description: 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.

Header file: <wchar.h>

Return values: Number of characters at the beginning of the **s1** string that are not included in the **s2** string

Parameters:

s1	Pointer to the string to be checked
s2	Pointer to the string used to check **s1**

Example:

```
#include <wchar.h>
const wchar_t *s1, *s2;
size_t ret;
    ret=wcscspn(s1,s2);
```

Remarks: The **wcscspn** function is the wide-character version of the **strcspn** function.

wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2) First Occurrence of Specified Characters

Description: Searches a specified string for the first occurrence of the character that is included in another string specified.

Header file: <wchar.h>

Return values: If the character is found: Pointer to the found character
If the character is not found: **NULL**

Parameters:

s1	Pointer to the string to be searched
s2	Pointer to the string that indicates the characters to search **s1** for

Example:

```
#include <wchar.h>
const wchar_t *s1, *s2;
char *ret;
    ret=wcspbrk(s1,s2);
```

Remarks: The **wcspbrk** function is the wide-character version of the **strpbrk** function.

wchar_t *wcsrchr(const wchar_t *s, wchar_t c) Last Occurrence of Specified Character

Description: Searches a specified string for the last occurrence of a specified character.

Header file: <wchar.h>

Return values: If the character is found: Pointer to the found character
If the character is not found: **NULL**

Parameters:

s	Pointer to the string to be searched
c	Character to search for

Example:

```
#include <wchar.h>
const wchar_t *s;
```

```
int c;
wchar_t *ret;
    ret=wcsrchr(s,c);
```

size_t wcsspn(const wchar_t *s1, const wchar_t *s2) Number of Consecutive Characters Specified

Description: Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are included in another string specified.

Header file: <wchar.h>

Return values: Number of characters at the beginning of the **s1** string that are included in the **s2** string

Parameters:

s1	Pointer to the string to be checked
s2	Pointer to the string used to check s1

Example:

```
#include <wchar.h>
const wchar_t *s1, *s2;
size_t ret;
    ret=wcsspn(s1,s2);
```

Remarks: The **wcsspn** function is the wide-character version of the **strspn** function.

wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2) First Occurrence of String

Description: Searches a specified string for the first occurrence of another string specified.

Header file: <wchar.h>

Return values: If the string is found: Pointer to the found string
If the string is not found: **NULL**

Parameters:

s1	Pointer to the string to be searched
s2	Pointer to the string to search for

Example:
```
#include <wchar.h>
const wchar_t *s1, *s2;
wchar_t *ret;
    ret=wcsstr(s1,s2);
```

wchar_t* wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t ** restrict ptr) Division into Tokens

Description: Divides a specified string into some tokens.

Header file: <wchar.h>

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
ptr	Pointer to the string where search is to be started at the next function call

Example:
```
#include <wchar.h>
static wchar_t s1[] = L"?a???b,,,#c";
static wchar_t s2[] = L"\t \t";
wchar_t *t, *p1, *p2;
t = wcstok(s1, L"?", &p1);      // t points to token L"a".
t = wcstok(NULL, L",", &p1);    // t points to token L"??b".
t = wcstok(s2, L" \t", &p2);  // t is a NULL pointer.
t = wcstok(NULL, L"#,", &p1); // t points to token L"c".
t = wcstok(NULL, L"?", &p1);    // t is a NULL pointer.
```

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.

wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n) Character Search in Storage Area

Description: Searches a specified storage area for the first occurrence of a specified character.

Header file: <wchar.h>

Return values: If the character is found: Pointer to the found character
If the character is not found: **NULL**

Parameters:

s	Pointer to the storage area to be searched
c	Character to search for
n	Number of characters to search

Example:

```
#include <wchar.h>
const wchar_t *s;
int c;
size_t n;
wchar_t *ret;
    ret=wmemchr(s,c,n);
```

Remarks: The **wmemchr** function is the wide-character version of the **memchr** function.

size_t wcslen(const wchar_t *s) Wide String Length

Description: Calculates the length of a wide string except the terminating null wide character.

Header file: <wchar.h>

Return values: Number of characters in the wide string

Parameters:

s	Pointer to the wide string to check the length of

Example:

```
#include <wchar.h>
const wchar_t *s;
size_t ret;
    ret=wcslen(s);
```

Remarks: The **wcslen** function is the wide-character version of the **strlen** function.

wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n) Character Repeating

Description: Sets a specified character a specified number of times at the beginning of a specified storage area.

Header file: <wchar.h>

Return values: Value of **s**

Parameters:

s	Pointer to storage area to set characters in
c	Character to be set
n	Number of characters to be set

Example:

```
#include <wchar.h>
wchar_t c, *s, *ret;
size_t n;
    ret=wmemset(s,c,n);
```

Remarks: The **wmemset** function is the wide-character version of the **memset** function.

long wctob(wint_t c) Wide Character Conversion to 1-Byte Representation

Description: Converts a wide character to 1-byte representation.

Header file: <stdio.h>, <wchar.h>

Return values: Normal: One-byte value converted from a wide character
Abnormal: **EOF**

Parameters: c Wide character

Example:
```
#include <stdio.h>
#include <wchar.h>
wint_t c;
int ret;
    ret=wctob(c);
```

Error conditions: If the wide character cannot be represented in one byte, **EOF** is returned.

Remarks: The **wctob** function checks if **c** is a member of the extended character set and corresponds to a multibyte character that can be represented in one byte in the initial shift state.

long mbsinit(const mbstate_t *ps) Conversion State Function

Description: Checks if a specified **mbstate_t** object indicates the initial conversion state.

Header file: <wchar.h>

Return values: Initial conversion state: Nonzero
Otherwise: 0

Parameters: ps Pointer to **mbstate_t** object

Example:
```
#include <wchar.h>
const mbstate_t *mt;
int ret;
    ret=mbsinit(mt);
```

**size_t mbrlen(const char * restrict s, size_t n,
mbstate_t *restrict ps)** Calculation of Bytes in Multibyte Character

Description: Calculates the number of bytes in a specified multibyte character.

Header file: <wchar.h>

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.

Parameters:

s	Pointer to multibyte string
n	Maximum number of bytes to be checked for multibyte character
ps	Pointer to **mbstate_t** object

Example:

```
#include <wchar.h>
const char *s;
size_t n;
const mbstate_t *mt;
int ret;
    ret=mbrlen(s, n, mt);
```

size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps) Multibyte Character Conversion to Wide Character

Description: Converts a multibyte character to a wide character.

Header file: <wchar.h>

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.

Parameters:

pwc	Pointer to wide string to store the obtained wide character
s	Pointer to multibyte string
n	Maximum number of bytes to be checked for multibyte character
ps	Pointer to **mbstate_t** object

Example:

```
#include <wchar.h>
wchar_t *pwc;
const char *s;
size_t n, ret;
mbstate_t *ps;
    ret=mbrtowc(pwc, s, n, ps);
```

Remarks: If an illegal multibyte sequence is detected, the **EILSEQ** macro value is set in **errno** and the conversion state is unspecified.

size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps) — Wide Character Conversion to Multibyte Character

Description: Converts a wide character to a multibyte character.

Header file: <wchar.h>

Return values:

Normal:	Number of bytes in the multibyte character
Abnormal:	(**size_t**)(–1): An illegal multibyte sequence is detected

Parameters:

s	Pointer to multibyte string
wc	Wide character to be converted
ps	Pointer to **mbstate_t** object

Example:

```
#include <wchar.h>
wchar_t wc;
char *s;
size_t ret;
mbstate_t *ps;
    ret=wctomb(s, wc, ps);
```

Error conditions: If an illegal multibyte sequence is detected, the **EILSEQ** macro value is set in **errno** and the conversion state is unspecified.

Remarks: The number of bytes in the multibyte character that is determined by the **wcrtomb** function includes shift sequences. The number of bytes never exceeds **MB_CUR_MAX**. When the conversion result is a null wide character, the initial conversion state is entered, but when necessary, a shift sequence is stored before the wide character to restore the initial state.

size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n) Multibyte String Conversion to Wide String

Description: Converts a multibyte string to a wide string.

Header file: <stdlib.h>

Return values:

Normal:	Number of characters written to wide string
Abnormal:	(**size_t**)(–1): An illegal multibyte sequence is detected.

Parameters:

wcs	Pointer to wide string
s	Pointer to multibyte string
n	Number of wide characters to be stored in wide string

Example:

```
#include <stdlib.h>
wchar_t *pwcs;
const char *s;
size_t n, ret;
    ret=mbstowcs(pwcs, s, n);
```

Remarks: The **mbstowcs** function converts a multibyte character sequence in the array indicated by **s**, which begins in the initial shift state, to a sequence of corresponding wide characters and stores **n** or fewer wide characters in the array indicated by **pwcs**.

When a null character is detected, it is converted to a null wide character and conversion is terminated. Each multibyte character is converted in the same way as an **mbtowc** function call, except that the conversion state of the **mbtowc** function is not affected. If copying between objects whose areas overlap is specified, the behavior is undefined.

Even a normal return value does not include the number of bytes of the terminating character.

When the return value is equal to **n**, the array is not terminated by a null character.

size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n) Wide String Conversion to Multibyte String

Description: Converts a wide string to a multibyte string.

Header file: <stdlib.h>

Return values:

Normal:	Number of bytes written to multibyte string
Abnormal:	(**size_t**)(–1): An illegal multibyte sequence is detected

Parameters:

s	Pointer to multibyte string
pwcs	Pointer to wide string
n	Number of bytes to be written to multibyte string

Example:

```
#include <stdlib.h>
const char *s;
wchar_t *pwcs;
size_t n, ret;
    ret=wcstombs(s,pwcs,n);
```

Remarks: The **wcstombs** function converts a wide character sequence in the array indicated by **pwcs** to a sequence of corresponding multibyte characters beginning in the initial state and stores them in the array indicated by **s**. Storing in the array is terminated when the number of multibyte characters exceeds the upper limit of **n** bytes or a null character is stored. Each wide character is converted in the same way as a **wctomb** function call, except that the conversion state of the **wctomb** function is not affected.

If copying between objects whose areas overlap is specified, the behavior is undefined.

Even a normal return value does not include the number of bytes of the terminating character.

When the return value is equal to **n**, the array is not terminated by a null character.

9.3.2 EC++ Class Libraries

(1) Overview of Libraries

This section describes the specifications of the EC++ class libraries, which can be used as standard libraries in C++ programs. The class library types and corresponding standard include files are described. The specifications of each class library are given in accordance with the library configuration.

- Library types
  Table 9.38 shows the class library types and the corresponding standard include files.

Table 9.38 Class Library Types and Corresponding Standard Include Files

Library Type	Description	Standard Include Files
Stream input/output class library	Performs input/output processing	<ios>, <streambuf>, <istream>, <ostream>, <iostream>, <iomanip>
Memory management library	Performs memory allocation and deallocation	<new>
Complex number calculation class library	Performs calculation of complex number data	<complex>
String manipulation class library	Performs string manipulation	<string>

(2) Stream Input/Output Class Library

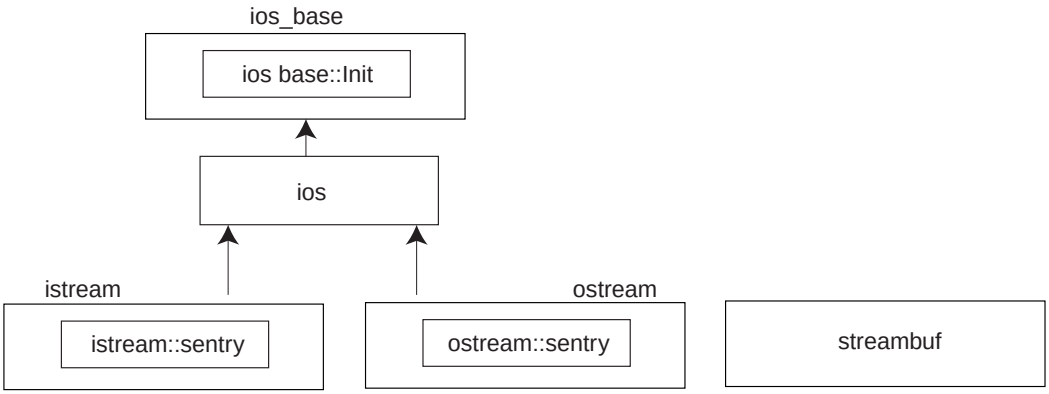
The header files for stream input/output class libraries are as follows:

- <ios>
  Defines data members and function members that specify input/output formats and manage the input/output states. The **<ios>** header file also defines the **Init** and **ios_base** classes in addition to the ios class.
- <streambuf>
  Defines functions for the stream buffer.
- <istream>
  Defines input functions from the input stream.
- <ostream>
  Defines output functions to the output stream.
- <iostream>

Defines input/output functions.

- <iomanip>
  Defines manipulators with parameters.

The following shows the inheritance relation of the above classes. An arrow (->) indicates that a derived class references a base class. The **streambuf** class has no inheritance relation.



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

1. ios_base::Init::Init()
   Constructor of class **Init**.
   Increments **init_cnt**.
2. ios_base::Init::~Init()
   Destructor of class **Init**.
   Decrements **init_cnt**.

(b) ios_base Class

Type	Definition Name	Description
Type	fmtflags	Type that indicates the format control information
	iostate	Type that indicates the stream buffer input/output state
	openmode	Type that indicates the open mode of the file
	seekdir	Type that indicates the seek state of the stream buffer
Variable	fmtfl	Format flag
	wide	Field width
	prec	Precision (number of decimal point digits) at output
	fillch	Fill character
Function	void _ec2p_init_base()	Initializes the base class
	void _ec2p_copy_base(ios_base&ios_base_dt)	Copies **ios_base_dt**
	ios_base()	Constructor
	~ios_base()	Destructor
	fmtflags flags() const	References the format flag (**fmtfl**)
	fmtflags flags(fmtflags fmtflg)	Sets **fmtflg**&format flag (**fmtfl**) to the format flag (**fmtfl**)
	fmtflags setf(fmtflags fmtflg)	Sets **fmtflg** to format flag (**fmtfl**)
	fmtflags setf(fmtflags fmtflg, fmtflags mask)	Sets **mask**&**fmtflg** to the format flag (**fmtfl**)
	void unsetf(fmtflags mask)	Sets ~**mask**&format flag (**fmtfl**) to the format flag (**fmtfl**)
	char fill() const	References the fill character (**fillch**)
	char fill(char ch)	Sets **ch** as the fill character (**fillch**)
	int precision() const	References the precision (**prec**)
	streamsize precision(streamsize preci)	Sets **preci** as precision (**prec**)
	streamsize width() const	References the field width (**wide**)
	streamsize width(streamsize wd)	Sets **wd** as field width (**wide**)

1. ios_base::fmtflags

   Defines the format control information relating to input/output processing.

   The definition for each bit mask of **fmtflags** is as follows:

```
const ios_base::fmtflags ios_base::boolalpha      = 0x0000;
const ios_base::fmtflags ios_base::skipws         = 0x0001;
const ios_base::fmtflags ios_base::unitbuf        = 0x0002;
const ios_base::fmtflags ios_base::uppercase      = 0x0004;
const ios_base::fmtflags ios_base::showbase       = 0x0008;
const ios_base::fmtflags ios_base::showpoint      = 0x0010;
const ios_base::fmtflags ios_base::showpos        = 0x0020;
const ios_base::fmtflags ios_base::left           = 0x0040;
const ios_base::fmtflags ios_base::right          = 0x0080;
const ios_base::fmtflags ios_base::internal       = 0x0100;
const ios_base::fmtflags ios_base::adjustfield    = 0x01c0;
const ios_base::fmtflags ios_base::dec            = 0x0200;
const ios_base::fmtflags ios_base::oct            = 0x0400;
const ios_base::fmtflags ios_base::hex            = 0x0800;
const ios_base::fmtflags ios_base::basefield      = 0x0e00;
const ios_base::fmtflags ios_base::scientific     = 0x1000;
const ios_base::fmtflags ios_base::fixed          = 0x2000;
const ios_base::fmtflags ios_base::floatfield     = 0x3000;
const ios_base::fmtflags ios_base::_fmtmask       = 0x3fff;
```

2. ios_base::iostate

   Defines the input/output state of the stream buffer.

   The definition for each bit mask of **iostate** is as follows:

```
const ios_base::iostate ios_base::goodbit         = 0x0;
const ios_base::iostate ios_base::eofbit          = 0x1;
const ios_base::iostate ios_base::failbit         = 0x2;
const ios_base::iostate ios_base::badbit          = 0x4;
const ios_base::iostate ios_base::_statemask      = 0x7;
```

3. ios_base::openmode
   Defines open mode of the file.
   The definition for each bit mask of **openmode** is as follows:

const ios_base::openmode ios_base::in	= 0x01;	Opens the input file.
const ios_base::openmode ios_base::out	= 0x02;	Opens the output file.
const ios_base::openmode ios_base::ate	= 0x04;	Seeks for **eof** only once after the file has been opened.
const ios_base::openmode ios_base::app	= 0x08;	Seeks for **eof** each time the file is written to.
const ios_base::openmode ios_base::trunc	= 0x10;	Opens the file in overwrite mode.
const ios_base::openmode ios_base::binary	= 0x20;	Opens the file in binary mode.

4. ios_base::seekdir
   Defines the seek state of the stream buffer.
   Determines the position in a stream to continue the input/output of data.
   The definition for each bit mask of **seekdir** is as follows:

```
const ios_base::seekdir ios_base::beg = 0x0;
const ios_base::seekdir ios_base::cur = 0x1;
const ios_base::seekdir ios_base::end = 0x2;
```

5. void ios_base::_ec2p_init_base()
   The initial settings are as follows:
   fmtfl = skipws | dec;
   wide = 0;
   prec = 6;
   fillch = ‘ ‘;

6. void ios_base::_ec2p_copy_base(ios_base& ios_base_dt)
   Copies **ios_base_dt**.

7. ios_base::ios_base()
   Constructor of class **ios_base**.
   Calls **Init::Init()**.

8. ios_base::~ios_base()

Destructor of class **ios_base**.

9. ios_base::fmtflags ios_base::flags() const
   References the format flag (**fmtfl**).
   Return value: Format flag (**fmtfl**)

10. ios_base::fmtflags ios_base::flags(fmtflags fmtflg)
    Sets **fmtflg**&format flag (**fmtfl**) to the format flag (**fmtfl**).
    Return value: Format flag (**fmtfl**) before setting

11. ios_base::fmtflags ios_base::setf(fmtflags fmtflg)
    Sets **fmtflg** to the format flag (**fmtfl**).
    Return value: Format flag (**fmtfl**) before setting

12. ios_base::fmtflags ios_base::setf((fmtflags fmtflg, fmtflags mask)
    Sets the **mask**&**fmtflg** value to the format flag (**fmtfl**).
    Return value: Format flag (**fmtfl**) before setting.

13. void ios_base::unsetf(fmtflags mask)
    Sets ~**mask**&format flag (**fmtfl**) to the format flag (**fmtfl**).

14. char ios_base::fill() const
    References the fill character (**fillch**).
    Return value: Fill character (**fillch**)

15. char ios_base::fill(char ch)
    Sets **ch** as the fill character (**fillch**).
    Return value: Fill character (**fillch**) before setting

16. int ios_base::precision() const
    References the precision (**prec**).
    Return value: Precision (**prec**)

17. streamsize ios_base::precision(streamsize preci)
    Sets **preci** as the precision (**prec**).
    Return value: Precision (**prec**) before setting

18. streamsize ios_base::width() const
    References the field width (**wide**).
    Return value: Field width (**wide**)

19. streamsize ios_base::width(streamsize wd)
    Sets **wd** as the field width (**wide**).
    Return value: Field width (**wide**) before setting

(c) ios Class

Type	Definition Name	Description
Variable	sb	Pointer to the **streambuf** object
	tiestr	Pointer to the **ostream** object
	state	State flag of **streambuf**
Function	ios()	Constructor
	ios(streambuf* 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**

1. ios::ios()
   Constructor of class **ios**.
   Calls **init(0)** and sets the initial value to the member object.

2. ios::ios(streambuf* sbptr)
   Constructor of class **ios**.
   Calls **init(sbptr)** and sets the initial value to the member object.

3. void ios::init(streambuf* sbptr)
   Sets **sbptr** to **sb**.
   Sets **state** and **tiestr** to 0.

4. virtual ios::~ios()
   Destructor of class **ios**.

5. 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**

6. bool ios::operator!() const
   Tests whether an error has been generated (**state**&(**badbit** | **failbit**)).
   Return value: An error has been generated: **true**
   No error has been generated: **false**

7. iostate ios::rdstate() const
   References the state flag (**state**).
   Return value: State flag (**state**)

8. void ios::clear(iostate st = goodbit)
   Clears the state flag (**state**) except for the specified state (**st**).
   If the pointer to the **streambuf** object (**sb**) is 0, **badbit** is set to the state flag (**state**).

9. void ios::setstate(iostate st)
   Sets st to the state flag (**state**).

10. bool ios::good() const
    Tests whether an error has been generated (**state**==**goodbit**).
    Return value: An error has been generated: **false**
    No error has been generated: **true**

11. bool ios::eof() const
    Tests for the end of the input stream (**state**&**eofbit**).
    Return value: End of the input stream has been reached: **true**
    End of the input stream has not been reached: **false**

12. bool ios::bad() const
    Tests whether an error has been generated (**state**&**badbit**).
    Return value: An error has been generated: **true**
    No error has been generated: **false**

13. bool ios::fail() const
    Tests whether the input text matches the requested pattern (**state**&(**badbit** | **failbit**)).
    Return value: Does not match the requested pattern: **true**
    Matches the requested pattern: **false**

14. ostream* ios::tie() const
    References the pointer (**tiestr**) to the **ostream** object.
    Return value: Pointer to the **ostream** object (**tiestr**)

15. 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

16. streambuf* ios::rdbuf() const
    References the pointer to the **streambuf** object (**sb**).
    Return value: Pointer to the **streambuf** object (**sb**)

17. 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

18. 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

1. 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**

2. ios_base& noshowbase(ios_base& str)

   Clears the output mode of prefixing a radix at the beginning of data.

Return value: **str**

3. 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**

4. ios_base& noshowpoint(ios_base& str)
   Clears the output mode of showing the decimal point.
   Return value: **str**

5. ios_base& showpos(ios_base& str)
   Specifies the output mode of generating the + sign (adds a + sign to a positive number).
   Return value: **str**

6. ios_base& noshowpos(ios_base& str)
   Clears the output mode of generating the + sign.
   Return value: **str**

7. ios_base& skipws(ios_base& str)
   Specifies the input mode of skipping spaces (skips consecutive spaces).
   Return value: **str**

8. ios_base& noskipws(ios_base& str)
   Clears the input mode of skipping spaces.
   Return value: **str**

9. 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**

10. ios_base& nouppercase(ios_base& str)
    Clears the output mode of converting letters to uppercases.

Return value: **str**

11. 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**

12. 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**

13. 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**

14. ios_base& dec(ios_base& str)
    Specifies the conversion radix to the decimal mode.
    Return value: **str**

15. ios_base& hex(ios_base& str)
    Specifies the conversion radix to the hexadecimal mode.
    Return value: **str**

16. ios_base& oct(ios_base& str)
    Specifies the conversion radix to the octal mode.
    Return value: **str**

17. ios_base& fixed(ios_base& str)
    Specifies the fixed-point output mode.
    Return value: **str**

18. 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
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].

Type	Definition Name	Description
Function	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
	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

Type	Definition Name	Description
Function	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

Note: 1. This class does not define the processing.

1. streambuf::streambuf()

   Constructor.

   The initial settings are as follows:

   _B_cnt_ptr = B_beg_ptr = B_next_ptr = B_end_ptr = C_flg_ptr = _B_len_ptr = 0

   B_beg_pptr = &B_beg_ptr

   B_next_pptr = &B_next_ptr

2. virtual streambuf::~streambuf()

Destructor.

3. streambuf* streambuf::pubsetbuf(char* s, streamsize n)
   Allocates the buffer for stream input/output.
   This function calls **setbuf (s,n)**.
   Return value: ***this**
4. pos_type streambuf::pubseekoff(off_type off, ios_base::seekdir way,
   ios_base::openmode which = (ios_base::openmode)(ios_base::in | ios_base::out))
   Moves the read or write position for the input/output stream by using the method specified by **way**.
   This function calls **seekoff(off,way,which)**.
   Return value: The stream position newly specified

5. pos_type streambuf::pubseekpos(pos_type sp, ios_base::openmode which =
   (ios_base::openmode)(ios_base::in | ios_base::out))
   Calculates the offset from the beginning of the stream to the current position.
   Moves the current stream pointer by the amount specified by **sp**.
   This function calls **seekpos(sp,which)**.
   Return value: The offset from the beginning of the stream

6. int streambuf::pubsync()
   Flushes the output stream.
   This function calls **sync()**.
   Return value: 0

7. streamsize streambuf::in_avail()
   Calculates the offset from the end of the input stream to the current position.
   Return value: If the position where data is read is valid: The offset from the end of the stream to the current position
   If the position where data is read is invalid: 0 (**showmanyc()** is called)

8. int_type streambuf::snextc()
   Reads one character. If the character read is not **eof**, the next character is read.
   Return value: If the character read is not **eof**: The character read
   If the character read is **eof**: **eof**

9. int_type streambuf::sbumpc()

   Reads one character and moves forward the pointer to the next.

   Return value: If the position where data is read is valid: The character read
   If the position where data is read is invalid: **eof**

10. int_type streambuf::sgetc()

    Reads one character.

    Return value: If the position where data is read is valid: The character read
    If the position where data is read is invalid: **eof**

11. int streambuf::sgetn(char* s, streamsize n)

    Sets **n** characters in the memory area specified by **s**. If an **eof** is found in the string read, setting is stopped.

    Return value: The specified number of characters

12. int_type streambuf::sputbackc(char c)

    If the data read position is correct and the put back data of the position is the same as **c**, the read position is put back.

    Return value: If the read position was put back: The value of **c**
    If the read position was not put back: **eof**

13. int streambuf::sungetc()

    If the data read position is correct, the read position is put back.

    Return value: If the read position was put back: The value that was put back
    If the read position was not put back: **eof**

14. int streambuf::sputc(char c)

    Inserts character **c**.

    Return value: If the write position is correct: The value of **c**
    If the write position is incorrect: **eof**

15. int_type streambuf::sputn(const char* s, streamsize n)

    Inserts **n** characters at the position specified by **s**.

    If the buffer is smaller than **n**, the number of characters for the buffer is inserted.

    Return value: The number of characters inserted

16. char* streambuf::eback() const

Calculates the start pointer of the input stream.
Return value: Start pointer

17. char* streambuf::gptr() const
    Calculates the next pointer of the input stream.
    Return value: Next pointer

18. char* streambuf::egptr() const
    Calculates the end pointer of the input stream.
    Return value: End pointer

19. void streambuf::gbump(int n)
    Moves forward the next pointer of the input stream by the amount specified by **n**.

20. void streambuf::setg(char* 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;

21. char* streambuf::pbase() const
    Calculates the start pointer of the output stream.
    Return value: Start pointer

22. char* streambuf::pptr() const
    Calculates the next pointer of the output stream.
    Return value: Next pointer

23. char* streambuf::epptr() const
    Calculates the end pointer of the output stream.
    Return value: End pointer

24. void streambuf::pbump(int n)
    Moves forward the next pointer of the output stream by the amount specified by **n**.

25. void streambuf::setp(char* 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;

26. 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.)

27. virtual pos_type streambuf::seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode = (ios_base::openmode)(ios_base::in | ios_base::out))

    Changes the stream position.

    Return value: -1 (This class does not define the processing.)

28. virtual pos_type streambuf::seekpos(pos_type sp, ios_base::openmode =
    (ios_base::openmode)(ios_base::in | ios_base::out))

    Changes the stream position.

    Return value: -1 (This class does not define the processing.)

29. virtual int streambuf::sync()

    Flushes the output stream.

    Return value: 0 (This class does not define the processing.)

30. virtual int streambuf::showmanyc()

    Calculates the number of valid characters in the input stream.

    Return value: 0 (This class does not define the processing.)

31. virtual streamsize streambuf::xsgetn(char* s, streamsize n)

    Sets **n** characters in the memory area specified by **s**.

    If the buffer is smaller than **n**, the number of characters for the buffer is inserted.

    Return value: The number of characters input

32. virtual int_type streambuf::underflow()
    Reads one character without moving the stream position.
    Return value: **eof** (This class does not define the processing.)

33. virtual int_type streambuf::uflow()
    Reads one character of the next pointer.
    Return value: **eof** (This class does not define the processing.)

34. virtual int_type streambuf::pbackfail(int_type c = eof)
    Puts back the character specified by **c**.
    Return value: **eof** (This class does not define the processing.)

35. virtual streamsize streambuf::xsputn(const char* s, streamsize n)
    Inserts **n** characters specified by **s** in to the stream position.
    If the buffer is smaller than **n**, the number of characters for the buffer is inserted.
    Return value: The number of characters inserted

36. virtual int_type streambuf::overflow(int_type c = eof)
    Inserts character **c** in the output stream.
    Return value: **eof** (This class does not define the processing.)

(f) istream::sentry Class

Type	Definition Name	Description
Variable	ok_	Whether the current state is input-enabled
Function	sentry(istream& is, bool noskipws = false)	Constructor
	~sentry()	Destructor
	operator bool()	References **ok**_

1. 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.

2. istream::sentry::~sentry()
   Destructor of internal class **sentry**.

3. istream::sentry::operator bool()
   References **ok_**.
   Return value: **ok_**

(g) istream Class

Type	Definition Name	Description
Variable	chcount	The number of characters extracted by the input function called last
Function	int _ec2p_getistr(char* 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

Type	Definition Name	Description
Function	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)	
	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)	

Type	Definition Name	Description
Function	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)	
	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::pubseekoff(0,cur,in)**.
	istream& seekg(pos_type pos)	Moves the current stream pointer by the amount specified by **pos**. This function calls **streambuf::pubseekpos(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::pubseekoff(off,dir)**.

1. int istream::_ec2p_getistr(char* str, unsigned int dig, int mode)
   Converts **str** to the radix specified by **dig**.
   Return value: The converted radix

2. istream::istream(streambuf* sb)
   Constructor of class **istream**.
   Calls **ios::init(sb)**.
   Specifies **chcount**=0.

3. virtual istream::~istream()
   Destructor of class **istream**.

4. istream& istream::operator>>(bool& n)
   istream& istream::operator>>(short& n)
   istream& istream::operator>>(unsigned short& n)
   istream& istream::operator>>(int& n)
   istream& istream::operator>>(unsigned int& n)
   istream& istream::operator>>(long& n)
   istream& istream::operator>>(unsigned long& n)
   istream& istream::operator>>(long long& n)
   istream& istream::operator>>(unsigned long long& n)
   istream& istream::operator>>(float& n)
   istream& istream::operator>>(double& n)
   istream& istream::operator>>(long double& n)
   Stores the extracted characters in **n**.
   Return value: ***this**

5. 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**

6. 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**

7. streamsize istream::gcount() const
   References **chcount** (number of extracted characters).
   Return value: **chcount**
8. int_type istream::get()
   Extracts characters.
   Return value: If characters are extracted: Extracted characters.
   If no characters are extracted: Calls **setstate(failbit)** and becomes **streambuf::eof**.

9. istream& istream::get(char& c)
   istream& istream::get(signed char& c)
   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**

10. istream& istream::get(char* s, streamsize n)
    istream& istream::get(signed char* s, streamsize n)
    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**

11. istream& istream::get(char* s, streamsize n, char delim)
    istream& istream::get(signed char* s, streamsize n, char delim)
    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**

12. 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**

13. 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**

14. istream& istream::getline(char* s, streamsize n)
    istream& istream::getline(signed char* s, streamsize n)
    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**

15. istream& istream::getline(char* s, streamsize n, char delim)
    istream& istream::getline(signed char* s, streamsize n, char delim)
    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**

16. 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**

17. int_type istream::peek()
    Seeks input characters that will be available next.
    Return value: If **ok_**==**false**: **streambuf::eof**
    If **ok_**!=**false**: **rdbuf()->sgetc()**

18. istream& istream::read(char* s, streamsize n)
    istream& istream::read(signed char* s, streamsize n)
    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**

19. streamsize istream::readsome(char* s, streamsize n)
    streamsize istream::readsome(signed char* s, streamsize n)
    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

20. 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**

21. 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**

22. int istream::sync()
    Checks for an input stream.
    This function calls **streambuf::pubsync()**.
    Return value: If there is no input stream: **streambuf::eof**
    If there is an input stream: 0

23. pos_type istream::tellg()
    Checks for the position of the input stream.
    This function calls **streambuf::pubseekoff(0,cur,in)**.
    Return value: Offset from the beginning of the stream
    If an error occurs during the input processing, -1 is returned.

24. istream& istream::seekg(pos_type pos)
    Moves the current stream pointer by the amount specified by **pos**.
    This function calls **streambuf::pubseekpos(pos)**.
    Return value: ***this**

25. istream& 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

1. 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)	

1. istream& operator>>(istream& in, char* s)
istream& operator>>(istream& in, signed char* s)
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**

2. istream& operator>>(istream& in, char& c)
   istream& operator>>(istream& in, signed char& c)
   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_**

1. 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**.

2. ostream::sentry::~sentry()
   Destructor of internal class **sentry**.
   If (**_ _ec2p_os->flags()** & **ios_base::unitbuf**) is true, **flush()** is called.

3. ostream::sentry::operator bool()
   References **ok_**.
   Return value: **ok_**

(k) ostream Class

Type	Definition Name	Description
Function	ostream(streambuf* 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()**.

Type	Definition Name	Description
Function	pos_type tellp()	Calculates the current write position. This function calls **streambuf::pubseekoff(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::pubseekpos(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::pubseekoff(off,dir)**.

1. ostream::ostream(streambuf* sbptr)
   Constructor.
   Calls **ios(sbptr)**.

2. virtual ostream::~ostream()
   Destructor.

3. ostream& ostream::operator<<(bool n)
   ostream& ostream::operator<<(short n)
   ostream& ostream::operator<<(unsigned short n)
   ostream& ostream::operator<<(int n)
   ostream& ostream::operator<<(unsigned int n)
   ostream& ostream::operator<<(long n)
   ostream& ostream::operator<<(unsigned long n)
   ostream& ostream::operator<<(long long n)
   ostream& ostream::operator<<(unsigned long long n)
   ostream& ostream::operator<<(float n)
   ostream& ostream::operator<<(double n)
   ostream& ostream::operator<<(long double n)
   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**

4. 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**

5. 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**

6. ostream& ostream::write(const char* s, streamsize n)
   ostream& ostream::write(const signed char* s, streamsize n)
   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**

7. ostream& ostream::flush()

   Flushes the output stream.

   This function calls **streambuf::pubsync()**.

   Return value: ***this**

8. pos_type ostream::tellp()

   Calculates the current write position.

   This function calls **streambuf::pubseekoff(0,cur,out)**.

   Return value: The current stream position

   If an error occurs during processing, -1 is returned.

9. ostream& ostream::seekp(pos_type pos)

   If no error occurs, the offset from the beginning of the stream to the current position is calculated.

   Moves the current stream pointer by the amount specified by **pos**.

   This function calls **streambuf::pubseekpos(pos)**.

Return value: ***this**

10. ostream& 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

1. ostream& endl(ostream& os)
   Inserts a new line code and flushes the output stream.
   This function calls **flush()**.
   Return value: **os**

2. ostream& ends(ostream& os)
   Inserts a **NULL** code into the output line.
   Return value: **os**

3. 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)	

1. ostream& operator<<(ostream& os, char s)
   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)
   If (**sentry::ok_**==**true**) and an error does not occur, **s** is inserted into the output stream. Otherwise **failbit** is set.
   Return value: **os**

(n) smanip Class Manipulator

Type	Definition Name	Description
Function	smanip resetiosflags(ios_base::fmtflags mask)	Clears the flag specified by the **mask** value
	smanip setiosflags(ios_base::fmtflags mask)	Specifies the format flag (**fmtfl**)
	smanip setbase(int base)	Specifies the radix used at output
	smanip setfill(char c)	Specifies the fill character (**fillch**)
	smanip setprecision(int n)	Specifies the precision (**prec**)
	smanip setw(int n)	Specifies the field width (**wide**)

1. smanip resetiosflags(ios_base::fmtflags mask)
   Clears the flag specified by the **mask** value.
   Return value: Target object of input/output

2. smanip setiosflags(ios_base::fmtflags mask)

Specifies the format flag (**fmtfl**).
Return value: Target object of input/output

3. smanip setbase(int base)
   Specifies the radix used at output.
   Return value: Target object of input/output

4. smanip setfill(char c)
   Specifies the fill character (**fillch**).
   Return value: Target object of input/output

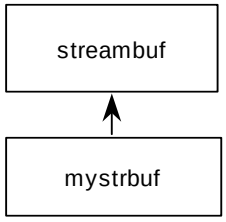
5. smanip setprecision(int n)
   Specifies the precision (**prec**).
   Return value: Target object of input/output

6. smanip setw(int n)
   Specifies the field width (**wide**).
   Return value: Target object of input/output

(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;
}
```

(3) Memory Management Library

The header file for the memory management library is as follows:

- <new>
  Defines the memory allocation/deallocation function.

By setting an exception handling function address to the **_ec2p_new_handler** variable, exception handling can be executed if memory allocation fails. The **_ec2p_new_handler** is a **static** variable and the initial value is **NULL**. If this handler is used, reentrance will be lost.

Operations required for the exception handling function:

- Creates an allocatable area and returns the area.
- Operations are not prescribed for cases where an area cannot be created.

Type	Definition Name	Description
Type	new_handler	Pointer type to the function that returns a **void** type
Variable	_ec2p_new_handler	Pointer to an exception handling function
Function	void* 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**

1. 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**

2. 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**

3. void* operator new(size_t size, void* ptr)
   Allocates the area specified by **ptr** as the storage area.
   Return value: **ptr**

4. void* operator new[ ](size_t size, void* ptr)
   Allocates the area specified by **ptr** as the array area.
   Return value: **ptr**

5. void operator delete(void* ptr)
   Deallocates the storage area specified by **ptr**.
   If **ptr** is **NULL**, no operation will be performed.

6. void operator delete[ ](void* ptr)
   Deallocates the array area specified by **ptr**.
   If **ptr** is **NULL**, no operation will be performed.

7. new_handler set_new_handler(new_handler new_P)
   Sets **new_P** to **_ec2p_new_handler**.
   Return value: **_ec2p_new_handler**

(4) Complex Number Calculation Class Library

The header file for the complex number calculation class library is as follows:

- <complex>
  Defines the **float_complex** and **double_complex** classes.

These classes have no derivation.

(a) float_complex Class

Type	Definition Name	Description
Type	value_type	**float** type
Variable	_re	Defines the real part of **float** precision
	_im	Defines the imaginary part of **float** precision
Function	float_complex(float re = 0.0f, float im = 0.0f)	Constructor
	float_complex(const double_complex& 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**.

1. float_complex::float_complex(float re = 0.0f, float im = 0.0f)

   Constructor of class **float_complex**.

   The initial settings are as follows:

   _re = re;

   _im = im;

2. float_ complex::float_complex(const double_complex& rhs)
   Constructor of class **float_complex**.
   The initial settings are as follows:
   _re = (float)rhs.real();
   _im = (float)rhs.imag();

3. float float_complex::real() const
   Acquires the real part.
   Return value: **this->_re**

4. float float_complex::imag() const
   Acquires the imaginary part.
   Return value: **this->_im**

5. float_complex& float_complex::operator=(float rhs)
   Copies **rhs** to the real part (**_re**).
   0.0f is assigned to the imaginary part (**_im**).
   Return value: ***this**

6. float_complex& float_complex::operator+=(float rhs)
   Adds **rhs** to the real part (**_re**) and stores the result in the real part (**_re**).
   The value of the imaginary part (**_im**) does not change.
   Return value: ***this**

7. float_complex& float_complex::operator-=(float rhs)
   Subtracts **rhs** from the real part (**_re**) and stores the result in the real part (**_re**).
   The value of the imaginary part (**_im**) does not change.
   Return value: ***this**

8. float_complex& float_complex::operator*=(float rhs)
   Multiplies ***this** by **rhs** and stores the result in ***this**.
   (_re=_re*rhs, _im=_im*rhs)
   Return value: ***this**

9. float_complex& float_complex::operator/=(float rhs)

Divides ***this** by **rhs** and stores the result in ***this**.
(_re=_re/rhs, _im=_im/rhs)
Return value: ***this**

10. float_complex& float_complex::operator=(const float_complex& rhs)
    Copies **rhs** to ***this**.
    Return value: ***this**

11. float_complex& float_complex::operator+=(const float_complex& rhs)
    Adds **rhs** to ***this** and stores the result in ***this**
    Return value: ***this**

12. float_complex& float_complex::operator-=(const float_complex& rhs)
    Subtracts **rhs** from ***this** and stores the result in ***this**.
    Return value: ***this**

13. float_complex& float_complex::operator*=(const float_complex& rhs)
    Multiplies ***this** by **rhs** and stores the result in ***this**.
    Return value: ***this**

14. 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**

Type	Definition Name	Description
Function	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)	
	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)	

Type	Definition Name	Description
Function	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
	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

Type	Definition Name	Description
Function	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

1. float_complex operator+(const float_complex& lhs)
   Performs unary + operation of **lhs**.
   Return value: **lhs**

2. float_complex operator+(const float_complex& lhs, const float_complex& rhs)
   float_complex operator+(const float_complex& lhs, const float& 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**

3. float_complex operator-(const float_complex& lhs)
   Performs unary - operation of **lhs**.
   Return value: **float_complex(-lhs.real(), -lhs.imag())**

4. float_complex operator-(const float_complex& lhs, const float_complex& rhs)
   float_complex operator-(const float_complex& lhs, const float& 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**

5. float_complex operator*(const float_complex& lhs, const float_complex& rhs)
   float_complex operator*(const float_complex& lhs, const float& 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**

6. float_complex operator/(const float_complex& lhs, const float_complex& rhs)
   float_complex operator/(const float_complex& lhs, const float& 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**

7. bool operator==(const float_complex& lhs, const float_complex& rhs)
   bool operator==(const float_complex& lhs, const float& rhs)
   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()**

8. bool operator!=(const float_complex& lhs, const float_complex& rhs)
   bool operator!=(const float_complex& lhs, const float& rhs)
   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()**

9. 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**

10. 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**

11. float real(const float_complex& x)
    Acquires the real part.
    Return value: **x.real()**

12. float imag(const float_complex& x)
    Acquires the imaginary part.
    Return value: **x.imag()**

13. float abs(const float_complex& x)
    Calculates the absolute value.
    Return value: $(|\mathbf{x.real()}|^2 + |\mathbf{x.imag()}|^2)^{1/2}$

14. float arg(const float_complex& x)
    Calculates the phase angle.
    Return value: **atan2f(x.imag(), x.real())**

15. float norm(const float_complex& x)
    Calculates the absolute value of the square.
    Return value: $|\mathbf{x.real()}|^2 + |\mathbf{x.imag()}|^2$

16. float_complex conj(const float_complex& x)
    Calculates the conjugate complex number.
    Return value: **float_complex(x.real(), (-1)*x.imag())**

17. 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))**

18. 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()))

19. float_complex cosh(const float_complex& x)

Calculates the complex hyperbolic cosine.

Return value: **cos(float_complex((-1)*x.imag(), x.real()))**

20. float_complex exp(const float_complex& x)

Calculates the exponent function.

Return value: **expf(x.real())*cosf(x.imag()),expf(x.real())*sinf(x.imag())**

21. float_complex log(const float_complex& x)

Calculates the natural logarithm (base e).

Return value: **float_complex(logf(abs(x)), arg(x))**

22. float_complex log10(const float_complex& x)

Calculates the common logarithm (base 10).

Return value: **float_complex(log10f(abs(x)), arg(x)/logf(10))**

23. float_complex pow(const float_complex& x, int y)
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)

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))**

24. 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()))**

25 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()))**

26. 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))**

27. float_complex tan(const float_complex& x)
    Calculates the complex tangent.
    Return value: **sin(x)/cos(x)**

28. 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**

1. double_complex::double_complex(double re = 0.0, double im = 0.0)
   Constructor of class **double_complex**.
   The initial settings are as follows:
   _re = re;
   _im = im;

2. double_complex::double_complex(const float_complex&)
   Constructor of class **double_complex**.
   The initial settings are as follows:
   _re = (double)rhs.real();
   _im = (double)rhs.imag();

3. double double_complex::real() const
   Acquires the real part.
   Return value: **this->_re**

4. double double_complex::imag() const
   Acquires the imaginary part.
   Return value: **this->_im**

5. double_complex& double_complex::operator=(double rhs)
   Copies **rhs** to the real part (**_re**).
   0.0 is assigned to the imaginary part (**_im**).
   Return value: ***this**

6. double_complex& double_complex::operator+=(double rhs)
   Adds **rhs** to the real part (**_re**) and stores the result in the real part (**_re**).
   The value of the imaginary part (**_im**) does not change.
   Return value: ***this**

7. double_complex& double_complex::operator-= (double rhs)
   Subtracts **rhs** from the real part (**_re**) and stores the result in the real part (**_re**).
   The value of the imaginary part (**_im**) does not change.
   Return value: ***this**

8. double_complex& double_complex::operator*=(double rhs)
   Multiplies ***this** by **rhs** and stores the result in ***this**.
   (_re=_re*rhs, _im=_im*rhs)
   Return value: ***this**

9. double_complex& double_complex::operator/=(double rhs)
   Divides ***this** by **rhs** and stores the result in ***this**.
   (_re=_re/rhs, _im=_im/rhs)
   Return value: ***this**

10. double_complex& double_complex::operator=(const double_complex& rhs)
    Copies **rhs** to ***this**.
    Return value: ***this**

11. double_complex& double_complex::operator+=(const double_complex& rhs)
    Adds **rhs** to ***this** and stores the result in ***this**.
    Return value: ***this**

12. double_complex& double_complex::operator-=(const double_complex& rhs)
    Subtracts **rhs** from ***this** and stores the result in ***this**.
    Return value: ***this**

13. double_complex& double_complex::operator*=(const double_complex& rhs)
    Multiplies ***this** by **rhs** and stores the result in ***this**.
    Return value: ***this**

14. 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)	
	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)	

Type	Definition Name	Description
Function	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

Type	Definition Name	Description
Function	double norm(const double_complex& x)	Calculates the absolute value of the square
	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

Type	Definition Name	Description
Function	double_complex tanh(const double_complex& x)	Calculates the complex hyperbolic tangent

1. double_complex operator+(const double_complex& lhs)
   Performs unary + operation of **lhs**.
   Return value: **lhs**

2. double_complex operator+(const double_complex& lhs, const double_complex& rhs)
   double_complex operator+(const double_complex& lhs, const double& 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**

3. double_complex operator-(const double_complex& lhs)
   Performs unary - operation of **lhs**.
   Return value: **double_complex(-lhs.real(), -lhs.imag())**

4. double_complex operator-(const double_complex& lhs, const double_complex& rhs)
   double_complex operator-(const double_complex& lhs, const double& 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**

5. double_complex operator*(const double_complex& lhs, const double_complex& rhs)
   double_complex operator*(const double_complex& lhs, const double& 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**

6. double_complex operator/(const double_complex& lhs, const double_complex& rhs)
   double_complex operator/(const double_complex& lhs, const double& 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**

7. bool operator==(const double_complex& lhs, const double_complex& rhs)
   bool operator==(const double_complex& lhs, const double& rhs)
   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()

8. bool operator!=(const double_complex& lhs, const double_complex& rhs)
   bool operator!=(const double_complex& lhs, const double& rhs)
   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()**

9. 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**

10. 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**

11. double real(const double_complex& x)
    Acquires the real part.
    Return value: **x.real()**

12. double imag(const double_complex& x)
    Acquires the imaginary part.
    Return value: **x.imag()**

13. double abs(const double_complex& x)
    Calculates the absolute value.

Return value: $(|\mathbf{x.real()}|^2 + |\mathbf{x.imag()}|^2)^{1/2}$

14. double arg(const double_complex& x)
    Calculates the phase angle.
    Return value: **atan2(x.imag(), x.real())**

15. double norm(const double_complex& x)
    Calculates the absolute value of the square.
    Return value: $|\mathbf{x.real()}|^2 + |\mathbf{x.imag()}|^2$

16. double_complex conj(const double_complex& x)
    Calculates the conjugate complex number.
    Return value: **double_complex(x.real(), (-1)*x.imag())**

17. 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))**

18. 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()))

19. double_complex cosh(const double_complex& x)
    Calculates the complex hyperbolic cosine.
    Return value: **cos(double_complex((-1)*x.imag(), x.real()))**

20. 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())**

21. double_complex log(const double_complex& x)
    Calculates the natural logarithm (base e).
    Return value: **double_complex(log(abs(x)), arg(x))**

22. double_complex log10(const double_complex& x)

Calculates the common logarithm (base 10).
Return value: **double_complex(log10(abs(x)), arg(x)/log(10))**

23. double_complex pow(const double_complex& x, int y)
    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)
    Calculates **x** to the **y**th power.
    If **pow(0,0)**, a domain error will occur.
    Return value: **exp(y*log(x))**

24. 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()))**

25. 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()))**

26. 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))**

27. double_complex tan(const double_complex& x)
    Calculates the complex tangent.
    Return value: **sin(x)/cos(x)**

28. double_complex tanh(const double_complex& x)
    Calculates the complex hyperbolic tangent.
    Return value: **sinh(x)/cosh(x)**

(5) String Handling Class Library

The header file for the string handling class library is as follows:

- <string>
  Defines class **string**.

This class has no derivation.

(a) string Class

Type	Definition Name	Description
Type	iterator	**char*** 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

Type	Definition Name	Description
Function	string(void)	Constructor
	string::string(const string& str, size_t pos = 0, size_t n = npos)	
	string::string(const char* str, size_t n)	
	string::string(const char* str)	
	string::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
	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)	

Type	Definition Name	Description
Function	string& 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**

Type	Definition Name	Description
Function	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**
	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**

Type	Definition Name	Description
Function	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**
	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**

Type	Definition Name	Description
Function	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**
	size_t find_last_of(char c, size_t pos = npos) const	Finds the position where character **c** appears most recently before position **pos**

Type	Definition Name	Description
Function	size_t find_first_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**

Type	Definition Name	Description
Function	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**

1. string::string(void)
   Sets as follows:
   s_ptr = 0;
   s_len = 0;
   s_res = 1;

2. 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**.

3. string::string(const char* str, size_t n)
   Sets as follows:
   s_ptr = **str**;
   s_len = **n**;
   s_res = **n** + 1;

4. string::string(const char* str)
   Sets as follows:
   s_ptr = **str**;
   s_len = length of string **str**;
   s_res = length of string **str** + 1;

5. string::string(size_t n, char c)
   Sets as follows:
   s_ptr = string of **n** characters, each of which is **c**
   s_len = **n**;
   s_res = **n** + 1;

6. string::~string()

Destructor of class **string**.
Deallocates the memory area where the string is stored.

7. string& string::operator(const string& str)
   Assigns the data of **str**.
   Return value: ***this**

8. string& string::operator=(const char* str)
   Creates a **string** object from **str** and assigns its data to the **string** object.
   Return value: ***this**

9. string& string::operator=(char c)
   Creates a **string** object from **c** and assigns its data to the **string** object.
   Return value: ***this**

10. string::iterator string::begin()
    string::const_iterator string::begin() const
    Calculates the start pointer of the string.
    Return value: Start pointer of the string

11. string::iterator string::end()
    string::const_iterator string::end() const
    Calculates the end pointer of the string.
    Return value: End pointer of the string

12. size_t string::size() const
    size_t string::length() const
    Calculates the length of the stored string.
    Return value: Length of the stored string

13. size_t string::max_size() const
    Calculates the size of the allocated memory area.
    Return value: Size of the allocated area

14. void string::resize(size_t n, char c)
    Changes the number of characters in the string that can be stored by the object to **n**.

If **n**<=**size()**, replaces the string with the original string with length **n**.

If **n**>**size()**, replaces the string with a string that has **c** appended to the end so that the length will be equal to **n**.

The length must be **n**<=**max_size()**.

If **n**>**max_size()**, the string length is **n**=**max_size()**.

15. void string::resize(size_t n)

Changes the number of characters in the string that can be stored by the object to **n**.

If **n**<=**size()**, replaces the string with the original string with length **n**.

The length must be **n**<=**max_size**.

16. size_t string::capacity() const

Calculates the size of the allocated memory area.

Return value: Size of the allocated memory area

17. void string::reserve(size_t res_arg = 0)

Re-allocates the memory area.

After **reserve()**, **capacity()** will be equal to or larger than the **reserve()** parameter.

When the memory area is re-allocated, all references, pointers, and **iterator** that references the elements of the numeric sequence become invalid.

18. void string::clear()

Clears the stored string.

19. bool string::empty() const

Checks whether the number of characters in the stored string is 0.

Return value: If the length of the stored string is 0: **true**

If the length of the stored string is not zero: **false**

20. const char& string::operator[ ](size_t pos) const

char& string::operator[ ](size_t pos)

const char& string::at(size_t pos) const

char& string::at(size_t pos)

References **s_ptr[pos]**.

Return value: If **n**< **s_len**: **s_ptr [pos]**

If **n**>= **s_len**: '\0'

21. string& string::operator+=(const string& str)
    Appends the string stored in **str** to the object.
    Return value: ***this**

22. string& string::operator+=(const char* str)
    Creates a **string** object from **str** and adds the string to the object.
    Return value: ***this**

23. string& string::operator+=(char c)
    Creates a **string** object from **c** and adds the string to the object.
    Return value: ***this**

24. string& string::append(const string& str)
    string& string::append(const char* str)
    Appends string **str** to the object.
    Return value: ***this**

25. 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**

26. string& string::append(const char* str, size_t n)
    Appends **n** characters of string **str** to the object.
    Return value: ***this**

27. string& string::append(size_t n, char c)
    Appends **n** characters, each of which is **c**, to the object.
    Return value: ***this**

28. string& string::assign(const string& str)
    string& string::assign(const char* str)
    Assigns string **str**.
    Return value: ***this**

29. string& string::assign(const string& str, size_t pos, size_t n)

Assigns **n** characters of string **str** to position **pos**.
Return value: ***this**

30. string& string::assign(const char* str, size_t n)
    Assigns **n** characters of string **str**.
    Return value: ***this**
31. string& string::assign(size_t n, char c)
    Assigns **n** characters, each of which is **c**.
    Return value: ***this**

32. string& string::insert(size_t pos1, const string& str)
    Inserts string **str** to position **pos1**.
    Return value: ***this**

33. 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**

34. string& string::insert(size_t pos, const char* str, size_t n)
    Inserts **n** characters of string **str** to position **pos**.
    Return value: ***this**

35. string& string::insert(size_t pos, const char* str)
    Inserts string **str** to position **pos**.
    Return value: ***this**

36. 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**

37. string::iterator string::insert(iterator p, char c = char())
    Inserts character **c** before the string specified by **p**.
    Return value: The inserted character

38. void string::insert(iterator p, size_t n, char c)
    Inserts **n** characters, each of which is **c**, before the character specified by **p**.

39. string& string::erase(size_t pos = 0, size_t n = npos)
    Deletes **n** characters starting from position **pos**.
    Return value: ***this**

40. iterator string::erase(iterator position)
    Deletes the character referenced by **position**.
    Return value: If the next **iterator** of the element to be deleted exists: The next **iterator** of the deleted element
    If the next **iterator** of the element to be deleted does not exist: **end()**

41. iterator string::erase(iterator first, iterator last)
    Deletes the characters in range [**first**, **last**].
    Return value: If the next **iterator** of **last** exists: The next **iterator** of **last**
    If the next **iterator** of **last** does not exist: **end()**

42. string& string::replace(size_t pos1, size_t n1, const string& str)
    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**

43. 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**

44. 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**

45. 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**

46. string& string::replace(iterator i1, iterator i2, const string& str)
    string& string::replace(iterator i1, iterator i2, const char* str)
    Replaces the string from position **i1** to **i2** with string **str**.
    Return value: ***this**

47. 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**

48. 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**

49. 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**

50. void string::swap(string& str)
    Swaps ***this** with string **str**.

51. const char* string::c_str() const
    const char* string::data() const
    References the pointer to the memory area where the string is stored.
    Return value: **s_ptr**

52. size_t string::find(const string& str, size_t pos = 0) const
    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

53. 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

54. size_t string::find(char c, size_t pos = 0) const
    Finds the position where character **c** first appears after position **pos**.

Return value: Offset of string

55. size_t string::rfind(const string& str, size_t pos = npos) const
    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
56. 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

57. size_t string::rfind(char c, size_t pos = npos) const
    Finds the position where character **c** appears most recently before position **pos**.
    Return value: Offset of string

58. size_t string::find_first_of(const string& str, size_t pos = 0) const
    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

59. 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

60. size_t string::find_first_of(char c, size_t pos = 0) const
    Finds the position where character **c** first appears after position **pos**.
    Return value: Offset of string

61. size_t string::find_last_of(const string& str, size_t pos = npos) const
    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

62. 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

63. size_t string::find_last_of(char c, size_t pos = npos) const

Finds the position where character **c** appears most recently before position **pos**.

Return value: Offset of string

64. size_t string::find_first_not_of(const string& str, size_t pos = 0) const
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

65. 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

66. size_t string::find_first_not_of(char c, size_t pos = 0) const

Finds the position where a character different from character **c** first appears after position **pos**.

Return value: Offset of string

67. size_t string::find_last_not_of(const string& str, size_t pos = npos) const
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

68. 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

69. size_t string::find_last_not_of(char c, size_t pos = npos) const

Finds the position where a character different from character **c** appears most recently before position **pos**.

Return value: Offset of string

70. string string::substr(size_t pos = 0, size_t n = npos) const

Creates an object from a string in the range [**pos**,**n**] of the stored string.

Return value: Object with a string in the range [**pos**,**n**]

71. int string::compare(const string& str) const

Compares the string with string **str**.

Return value: If the strings are the same: 0
If the strings are different: 1 when **this->s_len** > **str.s_len**,
-1 when **this->s_len** < **str.s_len**

72. int string::compare(size_t pos1, size_t n1, const string& str) const

Compares a string of **n1** characters starting from position **pos1** of ***this** with string **str**.

Return value: If the strings are the same: 0
If the strings are different: 1 when **this->s_len** > **str.s_len**,
-1 when **this->s_len** < **str.s_len**

73. int string::compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const

Compares a string of **n1** characters starting from position **pos1** with the string of **n2** characters from position **pos2** of string **str**.

Return value: If the strings are the same: 0
If the strings are different: 1 when **this->s_len** > **str.s_len**,
-1 when **this->s_len** < **str.s_len**

74. int string::compare(const char* str) const

Compares ***this** with string **str**.

Return value: If the strings are the same: 0
If the strings are different: 1 when **this->s_len** > **str.s_len**,
-1 when **this->s_len** < **str.s_len**

75. int string::compare(size_t pos1, size_t n1, const char* 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)	
	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)	

Type	Definition Name	Description
Function	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.

1. string operator+(const string& lhs, const string& rhs)
   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)

   Appends the string (characters) of **lhs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.

   Return value: Object where the linked strings are stored

2. bool operator==(const string& lhs, const string& rhs)
   bool operator==(const char* lhs, const string& rhs)
   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**

3. bool operator!=(const string& lhs, const string& rhs)
   bool operator!=(const char* lhs, const string& rhs)
   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**

4. bool operator<(const string& lhs, const string& rhs)
   bool operator<(const char* lhs, const string& rhs)
   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**

5. bool operator>(const string& lhs, const string& rhs)
   bool operator>(const char* lhs, const string& rhs)
   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**

6. bool operator<=(const string& lhs, const string& rhs)
   bool operator<=(const char* lhs, const string& rhs)
   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**

7. bool operator>=(const string& lhs, const string& rhs)
   bool operator>=(const char* lhs, const string& rhs)
   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**

8. void swap(string& lhs,string& rhs)
   Swaps the string of **lhs** with the string of **rhs**.

9. istream& operator>>(istream& is, string& str)
   Extracts a string to **str**.
   Return value: **is**

10. ostream& operator<<(ostream& os, const string& str)
    Inserts string **str**.
    Return value: **os**

11. 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**

12. 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**

9.3.3 Reentrant Library

A library generated by using the **reent** option of the standard library generator is able to execute all reentrants except for the **rand** and **srand** functions.

Table 9.39 lists libraries that are reentrant when the **reent** option is not specified. A function that is marked with Δ in the table sets the **errno** variable. Such a function can be assumed to be reentrant unless a program refers to **errno**.

Table 9.39 Reentrant Library List

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
stddef.h	offsetof	O	math.h	frexp	Δ
assert.h	assert	X		ldexp	Δ
ctype.h	isalnum	O		log	Δ
	isalpha	O		log10	Δ
	iscntrl	O		modf	Δ
	isdigit	O		pow	Δ
	isgraph	O		sqrt	Δ
	islower	O		ceil	Δ
	isprint	O		fabs	Δ
	ispunct	O		floor	Δ
	isspace	O		fmod	Δ
	isupper	O	mathf.h	acosf	Δ
	isxdigit	O		asinf	Δ
	tolower	O		atanf	Δ
	toupper	O		atan2f	Δ
math.h	acos	Δ		cosf	Δ
	asin	Δ		sinf	Δ
	atan	Δ		tanf	Δ
	atan2	Δ		coshf	Δ
	cos	Δ		sinhf	Δ
	sin	Δ		tanhf	Δ
	tan	Δ		expf	Δ
	cosh	Δ		frexpf	Δ
	sinh	Δ		ldexpf	Δ
	tanh	Δ		logf	Δ
	exp	Δ		log10f	Δ

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
mathf.h	modff	Δ	stdio.h	fputs	X
	powf	Δ		getc	X
	sqrtf	Δ		getchar	X
	ceilf	Δ		gets	X
	fabsf	Δ		putc	X
	floorf	Δ		putchar	X
	fmodf	Δ		puts	X
setjmp.h	setjmp	O		ungetc	X
	longjmp	O		fread	X
stdarg.h	va_start	O		fwrite	X
	va_arg	O		fseek	X
	va_end	O		ftell	X
stdio.h	fclose	X		rewind	X
	fflush	X		clearerr	X
	fopen	X		feof	X
	freopen	X		ferror	X
	setbuf	X		perror	X
	setvbuf	X	stdlib.h	atof	Δ
	fprintf	X		atoi	Δ
	fscanf	X		atol	Δ
	printf	X		atoll	Δ
	scanf	X		strtod	Δ
	sprintf	Δ		strtol	Δ
	sscanf	Δ		strtoul	Δ
	vfprintf	X		strtoll	Δ
	vprintf	X		strtoull	Δ
	vsprintf	Δ		rand	X
	fgetc	X		srand	X
	fgets	X		calloc	X
	fputc	X		free	X

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
stdlib.h	malloc	X	string.h	memcmp	O
	realloc	X		strcmp	O
	bsearch	O		strncmp	O
	qsort	O		memchr	O
	abs	O		strchr	O
	div	Δ		strcspn	O
string.h	labs	O		strpbrk	O
	llabs	O		strrchr	O
	ldiv	Δ		strspn	O
	lldiv	Δ		strstr	O
	memcpy	O		strtok	X
	strcpy	O		memset	O
	strncpy	O		strerror	O
	strcat	O		strlen	O
	strncat	O		memmove	O

Reentrant column: O: Reentrant
X: Non-reentrant
Δ: **errno** is set.

9.3.4 Unsupported Libraries

Table 9.40 lists the libraries which are specified in the C language specifications but not supported by this compiler.

Table 9.40 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, iswprintf, iswpunct, iswspace, iswupper, iswxdigit, iswctype, wctype, towlower, towupper, towctrans, wctrans
8	wchar.h	wcsftime, wcscoll, wcsxfrm, wctob, mbrtowc, wcrtomb, mbsrtowcs, wcsrtombs

Note: 1. The header file is not supported.

Section 10 Assembly Language Specifications

10.1 Coding Rules

10.1.1 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.

10.1.2 Names

Desired names can be defined and used in assembly-language files.

Names are classified into the following types.

Table 10.1 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.

10.1.3 Mnemonic Line Format

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.

10.1.4 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.

10.1.5 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 10.2 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 10.3 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 10.4 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

10.1.6 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): $1.17549435 \times 10^{-38}$ to $3.40282347 \times 10^{38}$
DOUBLE (64 bits): $2.2250738585072014 \times 10^{-308}$ to $1.7976931348623157 \times 10^{308}$

- 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) 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 10.5 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 10.6 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.
&	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 10.7 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 10.8 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 10.9 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	>, <, >=, <=, ==, !=

(3) 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. 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 10.10 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, BPSW, BPC, FINTV, PC, CPEN

  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.

(4) 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

(5) 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 10.11 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
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
```

10.1.7 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.
```

10.2 Optimum Instruction Selection

10.2.1 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 10.12 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".
2. The **#imm** range for **src** in the **INT** instruction is 0 to 255.
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 10.13 Instruction Formats of ADC and SBB Instructions

	Target of Optimum Selection			Code Size
Instruction Format	**src**	**src2**	**dest**	**[Bytes]**
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 10.14 Instruction Formats of ADD Instruction

	Target of Optimum Selection			Code Size
Instruction Format	**src**	**src2**	**dest**	**[Bytes]**
(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 10.15 Instruction Formats of AND, OR, SUB, and MUL Instructions

	Target of Optimum Selection			Code Size
Instruction Format	**src**	**src2**	**dest**	**[Bytes]**
AND/OR/SUB/MUL	#uimm:4	—	Rd	2
src,dest	#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 10.16 Instruction Formats of BMCnd Instruction

	Processing	Target of Optimum Selection			Code Size
Instruction Format	**Size**	**src**	**src2**	**dest**	**[Bytes]**
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 10.17 Instruction Formats of CMP Instruction

Instruction Format	Processing Size	Target of Optimum Selection			Code Size
		src	src2	dest	[Bytes]
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 10.18 Instruction Formats of DIV, DIVU, EMUL, EMULU, ITOF, MAX, MIN, TST, and XOR Instructions

	Target of Optimum Selection			Code Size
Instruction Format	**src**	**src2**	**dest**	**[Bytes]**
DIV/DIVU/	#simm:8	—	Rd	4
EMUL/EMULU/ITOF/	#simm:16	—	Rd	5
MAX/MIN/TST/XOR	#simm:24	—	Rd	6
src,dest	#imm:32	—	Rd	7
	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 10.19 Instruction Formats of FADD, FCMP, FDIV, FMUL, and FTOI Instructions

	Target of Optimum Selection			Code Size
Instruction Format	**src**	**src2**	**dest**	**[Bytes]**
FADD/FCMP/FDIV/	#imm:32	—	Rd	7
FMUL/FTOI	dsp:8[Rs].L	—	Rd	4
src,dest	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 10.20 Instruction Formats of MVTC, STNZ, and STZ Instructions

	Target of Optimum Selection			Code Size
Instruction Format	**src**	**src2**	**dest**	**[Bytes]**
MVTC/STNZ/STZ	#simm:8	—	Rd	4
src,dest	#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 10.21 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
	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

Instruction Format	size	Processing Size	Target of Optimum Selection			Code Size [Bytes]
			src	src2	dest	
	L	L	#simm:24	—	dsp:16[Rd]	7
MOV(.size) src,dest	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 10.22 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 10.23 Instruction Formats of PUSH Instruction

	Target of Optimum Selection			Code Size
Instruction Format	**src**	**src2**	**dest**	**[Bytes]**
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 10.24 Instruction Formats of ROUND Instruction

	Target of Optimum Selection			Code Size
Instruction Format	**src**	**src2**	**dest**	**[Bytes]**
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 10.25 Instruction Formats of SCCnd Instruction

		Target of Optimum Selection			Code Size
Instruction Format	**size**	**src**	**src2**	**dest**	**[Bytes]**
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 10.26 Instruction Formats of XCHG Instruction

Instruction Format	Processing Size	Target of Optimum Selection			Code Size
		src	src2	dest	[Bytes]
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 10.27 Instruction Formats of BCLR, BNOT, BSET, and BTST Instructions

Instruction Format	Processing Size	Target of Optimum Selection			Code Size
		src	src2	dest	[Bytes]
BCLR/BNOT/BSET/BTST	B	#imm:3	—	dsp:8[Rd].B	3
src,dest	B	#imm:3	—	dsp:16[Rd].B	4
	B	Rs	—	dsp:8[Rd].B	4
	B	Rs	—	dsp:16[Rd].B	5

10.2.2 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 10.1.5 (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 10.1.5 (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 10.28 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.

10.3 Assembler Directive Coding

The assembler directives are classified into general assembler directives (hereafter, simply called assembler directives) and assembler directives for high-level languages.

10.3.1 Address Directives

These directives control address specifications in the assembler.

The assembler handles relocatable address values except for the addresses in absolute-addressing sections.

Table 10.29 Address Directives

Directive	Function
.ORG	Declares the start address. The section including this directive becomes an absolute-addressing section.
.OFFSET	Specifies an offset from the beginning of the section. This directive can be used only in a relative-addressing section.
.ENDIAN	Specifies the endian for the section.
.BLKB	Allocates a RAM area in 1-byte units.
.BLKW	Allocates a RAM area in 2-byte units.
.BLKL	Allocates a RAM area in 4-byte units.
.BLKD	Allocates a RAM area in 8-byte units.
.BYTE	Stores 1-byte data in a ROM area.
.WORD	Stores 2-byte data in a ROM area.
.LWORD	Stores 4-byte data in a ROM area.
.FLOAT	Stores floating-point data represented by four bytes in a ROM area.
.DOUBLE	Stores floating-point data represented by eight bytes in a ROM area.
.ALIGN	Corrects a location counter to a multiple of the boundary alignment value.

.ORG Address Declaration

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:

```
.SECTION        value,ROMDATA
.ORG            0FF00H
.BYTE           "abcdefghijklmnopqrstuvwxyz"
.ORG            0FF80H
.BYTE           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
.END
```

The following example will generate an error because **.ORG** is not written immediately after **.SECTION**.

```
.SECTION        value,ROMDATA
.BYTE           "abcdefghijklmnopqrstuvwxyz"
.ORG            0FF80H
.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.

.OFFSET Offset Declaration

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:

```
.SECTION      value,ROMDATA
.BYTE         "abcdefghijklmnopqrstuvwxyz"
.OFFSET       80H
.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.

```
.SECTION      value,ROMDATA
.OFFSET       80H
.BYTE         "abcdefghijklmnopqrstuvwxyz"
.OFFSET       70H
.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.

.ENDIAN Endian Specification

Format: .ENDIAN∆BIG

.ENDIAN∆LITTLE

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:

```
.SECTION        value,ROMDATA
.ORG            0FF00H
.ENDIAN         BIG
.BYTE           "abcdefghijklmnopqrstuvwxyz"
```

The following example will generate an error because **.ENDIAN** is not written immediately after **.SECTION** or **.ORG**.

```
.SECTION        value,ROMDATA
.ORG            0FF00H
.BYTE           "abcdefghijklmnopqrstuvwxyz"
.ENDIAN         BIG
.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 1-Byte Area Allocation

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.

.BLKW 2-Byte Area Allocation

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.

.BLKL 4-Byte Area Allocation

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.

.BLKD 8-Byte Area Allocation

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.

.BYTE 1-Byte Data Storing

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 2-Byte Data Storing

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
.WORD         1
.WORD         symbol
.WORD         symbol+1
.WORD         1,2,3,4,5
.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 4-Byte Data Storing

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.

Examples:
```
.SECTION value,ROMDATA
.LWORD        1
.LWORD        symbol
.LWORD        symbol+1
.LWORD        1,2,3,4,5
.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 4-Byte Data Storing

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 8-Byte Data Storing

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 Address Correction

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.

10.3.2 Assembler Directives

These directives do not generate data corresponding to themselves but controls generation of machine code for instructions. They do not modify addresses.

Table 10.30 Assembler Directives

Directive	Function
.EQU	Defines a symbol.
.END	Specifies the end of an assembly-language file.
.INCLUDE	Inserts the contents of the specified file to the location where this directive is written.

.EQU Numeric Value Symbol Definition

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 Assembly-Language File End Declaration

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 Include File Specification

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.

10.3.3 Link Directives

These directives are used for relocatable assembly that enables a program to be written in multiple separate files.

Table 10.31 Link Directives

Directive	Function
.SECTION	Defines a section, which is the minimum unit used for address relocation.
.GLB	Declares an external symbol.
.RVECTOR	Registers a symbol as a variable vector.

.SECTION Section Definition

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:

```
.SECTION      program,CODE
NOP
.SECTION      ram,DATA
.BLKB         10
.SECTION      tbl1,ROMDATA
.BYTE         "abcd"
.SECTION      tbl2,ROMDATA,ALIGN=8
.LWORD        11111111H,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.

Notes: 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 Global Declaration

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 Variable Vector Registration

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.

Examples:

```
    .RVECTOR   50,_rvfunc
_rvfunc:
    MOV.L  #0,R1
    RTE
```

Remarks: Be sure to insert a space character or a tab between this directive and the operand.

10.3.4 Source List Directive

This directive controls the output information and format of the source list file. It does not affect code generation.

Table 10.32 Source List Directive

Directive	Function
.LIST	Controls whether to output information in assembly-language line units when generating a source list file.

.LIST Source List Output Control

Format: .LISTΔ[ON|OFF]

Description: This directive can stop (**OFF**) outputting lines to the source list file.

Even in the range where line output is stopped, error lines are output to the source list file.

This directive can start (**ON**) outputting lines to the source list file.

When this directive is not specified, all lines are output to the source 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.

10.3.5 Conditional Assembly Directives

These directives specify whether to assemble a specified range of lines.

Table 10.33 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 Conditional Assembly

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 10.34 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 10.1.6, 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.

10.3.6 Extended Function Directives

These directives do not affect code generation.

Table 10.35 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 Specified String Output

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.

? 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.

@ String Concatenation

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 Replacement with Source 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 Stack Value Definition for Specified Symbol

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 Line Number Change

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 Replacement Symbol Definition

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.

10.3.7 Macro Directives

These directives define macro functions and repeat macro functions.

Table 10.36 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 Macro Definition

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             .MACRO  p1,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 10.1.2, 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 differ from that of the parameters, the assembler outputs a warning message.

.EXITM Macro Expansion Termination

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           .MACRO  value
        .IF     value == 0
                .EXITM
        .ELSE
                .BLKB   value
        .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 Declaration of Local Label in Macro

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
  .LOCAL      m1     ; 'm1' is macro local label
m1:
  nop
  bra  m1
  .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 End of 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 Beginning of 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 End of 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 Replacement with Number of Macro Arguments

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.

```
        .GLB   mem
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 Replacement with Current Macro Repetition Count

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

        mac     3,R3    ; Macro call

        .MREPEAT   3    ; 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 Replacement with Length of Specified String

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 Replacement with Start Position of 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 String Extraction

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.

10.3.8 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 10.37 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.

Section 11 Compiler Error Messages

11.1 Error Format and Error Levels

This section gives a list of error messages and explains details of errors in the following format.

Error number **(Error level) Error message**
Error details

There are five different error levels, corresponding to different degrees of seriousness.

Error Level	Error Type	Description
(I)	Information	Processing is continued.
(W)	Warning	Processing is continued.
(E)	Error	Option analysis processing is continued; processing is interrupted.
(F)	Fatal	Processing is interrupted.
(–)	Internal	Processing is interrupted.

11.2 List of Messages

C0005 (I) Precision lost
Precision may be lost when assigning with type conversion a right hand side value to the left hand side value.

C0006 (I) Conversion in argument
A function parameter expression is converted into a parameter type specified in the prototype declaration.

C0008 (I) Conversion in return
A return statement expression is converted into a value type that should be returned from a function.

C0011 (I) Used before set symbol: "variable name" in "function name"
A local variable is used before setting its value.

C0101 (I) Optimizing range divided in function "function name"
The optimizing range of the function **function name** is divided into many sections.

C0102 (I) Register is not allocated to "variable name" in "function name"
Any register cannot be allocated to the variable of the **register** storage class.

C1026 (W) Address of packed member
The address of a structure member specified with **pack=1** is acquired.

C1300 (W) Command parameter specified twice
The same compiler option is specified more than once. Uses the last specified compiler option.

C1301 (W) "option" option ignored
option is ignored at compilation.

C1308 (W) Duplicate number specified in option "option": "number"
The same number is specified twice in **option**.

C1309 (W) Section name "SI" or "SU" specified
SI or **SU** is specified for **section name**. The compiled data is output with the specified section name.

C1315 (W) File_inline "file name" ignored by same file as source file
The file to be compiled is specified by the **file_inline** option. The **file_inline** option is ignored and compilation is continued.

C1316 (W) "target macro" is not a valid predefined macro name
Macro name <**macro name**> is not a predefined macro. The **undefine** option specification is ignored.

C1317 (W) "option 1" and "option 2" are specified
Both **option 1** and **option 2**, which have conflicting meanings, have been specified. Although both options are valid, check that the combination is intended.

C1402 (W) #pragma section ignored
The **#pragma section** specification is ignored.

C1410 (W) A struct/union/class has different pack specifications
A single structure, union, or class has members with different **pack** specifications.

C1600 (W) Debugging information describing location of "name" is lost
Symbol information on **name** was not output.

C1800 (W) Variable "variable name" type mismatch in files
The type of the variable indicated by "variable name" differs between files. Delete the **file_inline** option.

C1801 (W) Using "function item" at influence the code generation of "NC" compiler
The specified **function item** that affects the compatibility with the NC compiler is used.

C1802 (E) (W) Using "function item" at influence the code generation of "H8" compiler
The specified **function item** that affects the compatibility with the H8 compiler is used.

C1803 (W) Address taken "variable name". It may cause an upset endian indirect reference
The address of 8-byte variable "variable name" in the endian that does not match the **endian** option setting is acquired. The endian processing may cause an incorrect indirect reference.

C1804 (W) Using incompatible int type
As the **int_to_short** option is invalid during C++ compilation, the **int** type size differs between C++ compilation and C compilation. This message is output when an external name of a C program may be referred to by a C++ program.

C1805 (W) "symbol name" is not confirmed in ROM by map option
External reference symbol **symbol name** declared with the **const** qualifier was not confirmed as a symbol in ROM through the **map** option processing.

C1806 (W) "symbol name" is regarded in ROM by map section
External reference symbol **symbol nam**e declared without the **const** qualifier was regarded as a symbol in ROM through the **map** option processing.

C1807 (E)(W) Using “function item” at influence the code generation of "SuperH" compiler
The specified **function item** (such as an option or **#pragma**) that affects the compatibility with the SuperH compiler is used.

C1950 (W) Nothing to compile, assemble or link (input and output combination)
There is no code that should be compiled, assembled, or linked. Check the combination of the input file configuration and the **output** option specification. The arguments that are not processed are listed under **Ignored argument(s):**.

C2021 (E) Invalid number specified in option "option": "number"
An invalid value is specified in **option**. Check the range of the value.

C2022 (E) Error level message cannot be changed: "change_message"
The level of an error-level message cannot be changed.

C2023 (E) Same register is used at base option.
The same register is specified for multiple areas having different **base** option settings.

C2024 (E) Base register is already used at fint_register option.
The register that is disabled by the **fint_register** option is specified by the **base** option.

C2025 (E) Base option address constant overflow
An address outside the range from 0x00000000 to 0xffffffff is specified by the **base** option.

C2026 (E) Illegal register of base option
An illegal register number (other than **R8** to **R13**) is specified by the **base** option.

C2027 (E) Cannot read specified file "file name"
The specified file cannot be read correctly. Check the file specification.

C2028 (E) Base register conflicts with option "option name"
The register specified in the **base** option has already been specified and used in option **option name**.

C2203 (E) Illegal member reference for "."
The type of the expression on the left side of operator "." is neither a structure or a union.

C2240 (E) Illegal section naming
There is an error in section naming. The same section name is specified for different use of the section.

C2450 (E) Illegal #pragma option declaration
There is an error in a **#pragma** option declaration.

C2550 (E) Assignment of ROM section object "variable name"
Variable **variable name** in the ROM section was written to.
The **-rom** option might not have been applied correctly at linkage.

C2700 (E) Function "function name" in #pragma interrupt already declared
The function specified by **#pragma interrupt** (interrupt function declaration) has already been declared as a normal function.

C2701 (E) Multiple interrupt for one function
An interrupt function declaration **#pragma interrupt** has been declared more than once for the same function.

C2703 (E) Illegal #pragma interrupt declaration
The interrupt function declaration by **#pragma interrupt** is incorrect.

C2704 (E) Illegal reference to interrupt function
The interrupt function reference is illegal.

C2710 (E) Section name too long
The specified section name exceeds the limit.

C2711 (E) Section name table overflow
The number of specified sections exceeds the limit.

C2714 (E) Usable stack area overflow
An attempt was made to access the stack in an area that cannot be accessed in SP-relative addressing mode and instruction generation failed.
This error may be caused by a negative value specified for an index of an array or a too large auto variable area. Check the source code.

C2800 (E) Illegal parameter number in in-line function
The number of parameters to be used for an intrinsic function do not match.

C2801 (E) Illegal parameter type in in-line function
There are different parameter types in an intrinsic function.

C2802 (E) Parameter out of range in in-line function
A parameter exceeds the range that can be specified in an intrinsic function.

C2803 (E) Invalid offset value in in-line function
An argument for an intrinsic function is incorrectly specified.

C2804 (E) Illegal in-line function
The code has an intrinsic function that cannot be used with the specified **cpu** option.

C2806 (E) Multiple #pragma for one function
Multiple **#pragma** directives specified for a single function do not match each other.

C2831 (E) Multiple #pragma entry declaration
There are two or more **#pragma entry** declarations.

C2833 (E) Multiple #pragma stacksize declaration
There are multiple **#pragma stacksize** declarations with **si** or **su** specification.

C2854 (E) Illegal address in #pragma address
The specified address has either of the following errors.
(1) A single address is specified for different variables.
(2) The address ranges specified for different variables overlap each other.

C2860 (E) Missing #pragma oscall for "service call name"
There is no **#pragma oscall** specification that is necessary for function **service call name**.

C3009 (F) String literal too long
The number of characters in a string exceeds the limit. The number of bytes obtained by concatenating the strings specified continuously is counted as the number of characters; that is, the number of characters in a string is not the string length in the source program but the bytes contained in the string data including an escape sequence as one character.

C3019 (F) Cannot open source file "file name"
A source file cannot be opened.

C3020 (F) Source file input error "file name"
A source file or include file cannot be read.

C3021 (F) Memory overflow
The compiler cannot allocate sufficient memory to compile the program.

C3023 (F) Type nest too deep
The number of types (pointer, array, and function types) qualifying a basic type exceeds the limit.

C3024 (F) Array dimension too deep
The number of array dimensions exceeds the limit.

C3025 (F) Source file not found
A source file name is not specified in the command line.

C3030 (F) Too many compound statements
The number of compound statements in a single function exceeds the limit.

C3031 (F) Data size overflow
The size of an array or a structure exceeds the limit.

C3203 (F) Assembly source line too long
The assembly source line is too long to output.

C3204 (F) Illegal stack access
The size of a stack to be used in a function (including a local variable area, register save area, and parameter push area to call other functions) or a parameter area to call the function exceeds 2 Gbytes.

C3300 (F) Cannot open internal file
An error occurred due to one of the following three causes:

(1) An intermediate file internally generated by the compiler cannot be opened.

(2) A file that has the same file name as the intermediate file already exists.

(3) A file which the compiler uses internally cannot be opened.

C3301 (F) Cannot close internal file
An intermediate file internally generated by the compiler cannot be closed. Make sure the compiler is correctly installed.

C3302 (F) Cannot input internal file
An intermediate file internally generated by the compiler cannot be read. Make sure the compiler is correctly installed.

C3303 (F) Cannot output internal file
An intermediate file internally generated by the compiler cannot be written to. Increase the disk space.

C3304 (F) Cannot delete internal file
An intermediate file internally generated by the compiler cannot be deleted. Check that the intermediate file generated by the compiler is not being accessed.

C3305 (F) Invalid command parameter "option name"
An invalid compiler option is specified.

C3306 (F) Interrupt in compilation
An interrupt generated by a **(Ctrl)** + **C** command (from a standard input terminal) is detected during compilation.

C3307 (F) Compiler version mismatch
File versions in the compiler do not match the other file versions. Refer to the Install Guide for the installation procedure, and reinstall the compiler.

C3308 (F) Cannot create file "file name"
The compiler cannot create necessary files.

C3320 (F) Command parameter buffer overflow
The command line specification exceeds 4096 characters.

C3321 (F) Illegal environment variable
An error occurred due to one of the following four causes:

(1) The environment variable **BIN_RX** was not specified.

(2) An execution file path name of the compiler was not specified for the environment variable **BIN_RX**.

(3) A file name was specified incorrectly when the environment variable **BIN_RX** was specified or the number of characters in a path name exceeds the limit of 118 characters.

(4) A value other than **RX600** is specified for environment variable **CPU_RX**.

C3322 (F) Lacking cpu specification
The CPU type is not specified. Specify the CPU type by the **cpu** option or environment variable **CPU_RX**.

C3900 (E) Input file not found. – "file name"
The specified input file cannot be found.

C3901 (E) Input file read error. – "file name"
A read error occurred in the input file.

C3902 (E) Invalid file name. – "file name"
A character that is not allowed is specified in the input file name.

C3903 (E) Invalid option. – "option specification"
The option specification is not correct.

C3905 (E) Cannot build temporary file.
A temporary file cannot be created. Check if the compiler environment settings are correct.

C3906 (E) Memory overflow.
There is not sufficient memory for the compiler processing.

C3907 (E) Tool execute error.
Initiation of the compiler, assembler, or optimizing linkage editor has failed.

C3908 (E) Cannot delete temporary file.
The temporary file cannot be deleted. Check if the compiler environment settings are correct.

C4000-C4999 (—) Internal error
An internal error occurred during compilation. Report the error occurrence to your local Renesas sales office.

C5001 (E) Last line of file ends without a newline

C5002 (E) Last line of file ends with a backslash

C5003 (F) #include file "file name" includes itself

C5004 (F) Out of memory

C5005 (F) Could not open source file "name"

C5006 (E) Comment unclosed at end of file

C5007 (E) (I) Unrecognized token

C5008 (E) (I) Missing closing quote

C5009 (I) Nested comment is not allowed

C5010 (E) "#" not expected here

C5011 (E) (W) Unrecognized preprocessing directive

C5012 (E) (W) Parsing restarts here after previous syntax error

C5013 (E) (F) Expected a file name

C5014 (E) Extra text after expected end of preprocessing directive

C5016 (F) "name" is not a valid source file name

C5017 (E) Expected a "]"

C5018 (E) Expected a ")"

C5019 (E) Extra text after expected end of number

C5020 (E) Identifier "name" is undefined

C5021 (W) Type qualifiers are meaningless in this declaration

C5022 (E) Invalid hexadecimal number

C5023 (E) Integer constant is too large

C5024 (E) Invalid octal digit

C5025 (E) Quoted string should contain at least one character

C5026 (E) Too many characters in character constant

C5027 (W) Character value is out of range

C5028 (E) Expression must have a constant value

C5029 (E) Expected an expression

C5030 (E) Floating constant is out of range

C5031 (E) (W) Expression must have integral type

C5032 (E) Expression must have arithmetic type

C5033 (E) Expected a line number

C5034 (E) Invalid line number

C5035 (F) #error directive: "line number"

C5036 (E) The #if for this directive is missing

C5037 (E) The #endif for this directive is missing

C5038 (E) (W) Directive is not allowed -- an #else has already appeared

C5039 (E) (W) Division by zero

C5040 (E) Expected an identifier

C5041 (E) Expression must have arithmetic or pointer type

C5042 (E) (W) Operand types are incompatible ("type1" and "type2")

C5044 (E) Expression must have pointer type

C5045 (W) #undef may not be used on this predefined name

C5046 (W) "macro name" is predefined; attempted redefinition ignored

C5047 (W) Incompatible redefinition of macro "name" (declared at line "line number")

C5049 (E) Duplicate macro parameter name

C5050 (E) "##" may not be first in a macro definition

C5051 (E) "##" may not be last in a macro definition

C5052 (E) Expected a macro parameter name

C5053 (E) Expected a ":"

C5054 (W) Too few arguments in macro invocation

C5055 (W) Too many arguments in macro invocation

C5056 (E) Operand of sizeof may not be a function

C5057 (E) This operator is not allowed in a constant expression

C5058 (E) This operator is not allowed in a preprocessing expression

C5059 (E) Function call is not allowed in a constant expression

C5060 (E) This operator is not allowed in an integral constant expression

C5061 (W) Integer operation result is out of range

C5062 (W) Shift count is negative

C5063 (W) Shift count is too large

C5064 (W) Declaration does not declare anything

C5065 (E) (W)Expected a ";"

C5066 (E) Enumeration value is out of "int" range

C5067 (E) Expected a "}"

C5068 (W) Integer conversion resulted in a change of sign

C5069 (W) Integer conversion resulted in truncation

C5070 (E) Incomplete type is not allowed

C5071 (E) Operand of sizeof may not be a bit field

C5075 (E) Operand of "*" must be a pointer

C5076 (W) Argument to macro is empty

C5077 (E) This declaration has no storage class or type specifier

C5078 (E) A parameter declaration may not have an initializer

C5079 (E) Expected a type specifier

C5080 (E) (W) A storage class may not be specified here

C5081 (E) More than one storage class may not be specified

C5082 (W) Storage class is not first

C5083 (W) Type qualifier specified more than once

C5084 (E) Invalid combination of type specifiers

C5085 (W) Invalid storage class for a parameter

C5086 (E) Invalid storage class for a function

C5087 (E) A type specifier may not be used here

C5088 (E) Array of functions is not allowed

C5089 (E) Array of void is not allowed

C5090 (E) Function returning function is not allowed

C5091 (E) Function returning array is not allowed

C5092 (E) Identifier-list parameters may only be used in a function definition

C5093 (E) Function type may not come from a typedef

C5094 (E) The size of an array must be greater than zero

C5095 (E) Array is too large

C5096 (W) A translation unit must contain at least one declaration

C5097 (E) A function may not return a value of this type

C5098 (E) An array may not have elements of this type

C5099 (E) (W) A declaration here must declare a parameter

C5100 (E) Duplicate parameter name

C5101 (E) "name" has already been declared in the current scope

C5102 (E) Forward declaration of enum type is nonstandard

C5103 (E) Class is too large

C5104 (E) Struct or union is too large

C5105 (E) Invalid size for bit field

C5106 (E) Invalid type for a bit field

C5107 (E) (W) Zero-length bit field must be unnamed

C5108 (W) Signed bit field of length 1

C5109 (E) Expression must have (pointer-to-) function type

C5110 (E) Expected either a definition or a tag name

C5111 (W) Statement is unreachable

C5112 (E) Expected "while"

C5114 (E) (W) Entity-kind "name" was referenced but not defined

C5115 (E) A continue statement may only be used within a loop

C5116 (E) A break statement may only be used within a loop or switch

C5117 (W) Non-void entity-kind "name" should return a value

C5118 (E) A void function may not return a value

C5119 (E) Cast to type "type" is not allowed

C5120 (E) Return value type does not match the function type

C5121 (E) A case label may only be used within a switch

C5122 (E) A default label may only be used within a switch

C5123 (E) Case label value has already appeared in this switch

C5124 (E) Default label has already appeared in this switch

C5125 (E) Expected a "("

C5126 (E) Expression must be an lvalue

C5127 (E) Expected a statement

C5128 (W) Loop is not reachable from preceding code

C5129 (E) A block-scope function may only have extern storage class

C5130 (E) Expected a "{"

C5131 (E) Expression must have pointer-to-class type

C5132 (E) Expression must have pointer-to-struct-or-union type

C5133 (E) Expected a member name

C5134 (E) Expected a field name

C5135 (E) Entity-kind "name" has no member "member name"

C5136 (E) Entity-kind "name" has no field "field name"

C5137 (E) (W) Expression must be a modifiable lvalue

C5138 (E) (W) Taking the address of a register field is not allowed

C5139 (E) Taking the address of a bit field is not allowed

C5140 (E) (W) Too many arguments in function call

C5141 (E) Unnamed prototyped parameters not allowed when body is present

C5142 (E) Expression must have pointer-to-object type

C5143 (F) Program too large or complicated to compile

C5144 (E) A value of type "type1" cannot be used to initialize an entity of type "type2"

C5145 (E) Entity-kind "name" may not be initialized

C5146 (E) Too many initializer values

C5147 (E) (W) Declaration is incompatible with "name" (declared at line "line number")

C5148 (E) Entity-kind "name" has already been initialized

C5149 (E) A global-scope declaration may not have this storage class

C5150 (E) A type name may not be redeclared as a parameter

C5151 (E) A typedef name may not be redeclared as a parameter

C5152 (W) Conversion of nonzero integer to pointer

C5153 (E) Expression must have class type

C5154 (E) Expression must have struct or union type

C5155 (W) Old-fashioned assignment operator

C5156 (W) Old-fashioned initializer

C5157 (E) (W) Expression must be an integral constant expression

C5158 (E) Expression must be an lvalue or a function designator

C5159 (E) Declaration is incompatible with previous "name" (declared at line "line number")

C5160 (E) Name conflicts with previously used external name "name"

C5161 (W) Unrecognized #pragma

C5163 (F) Could not open temporary file "name"

C5164 (F) Name of directory for temporary files is too long ("name")

C5165 (E) Too few arguments in function call

C5166 (E) Invalid floating constant

C5167 (E) Argument of type "type1" is incompatible with parameter of type "type2"

C5168 (E) A function type is not allowed here

C5169 (E) (W) Expected a declaration

C5170 (W) Pointer points outside of underlying object

C5171 (E) Invalid type conversion

C5172 (W) (I) External/internal linkage conflict with previous declaration

C5173 (E) (W) Floating-point value does not fit in required integral type

C5174 (I) Expression has no effect

C5175 (E) (W) Subscript out of range

C5177 (W) Entity-kind "name" was declared but never referenced

C5178 (W) "&" applied to an array has no effect

C5179 (W) Right operand of "%" is zero

C5180 (W) (I) Argument is incompatible with formal parameter

C5181 (W) Argument is incompatible with corresponding format string conversion

C5182 (F) Could not open source file "name" (no directories in search list)

C5183 (E) Type of cast must be integral

C5184 (E) Type of cast must be arithmetic or pointer

C5185 (I) Dynamic initialization in unreachable code

C5186 (W) Pointless comparison of unsigned integer with zero

C5187 (I) Use of "=" where "==" may have been intended

C5188 (W) Enumerated type mixed with another type

C5189 (F) Error while writing "file name" file

C5190 (F) Invalid intermediate language file

C5191 (W) Type qualifier is meaningless on cast type

C5192 (W) Unrecognized character escape sequence

C5193 (I) Zero used for undefined preprocessing identifier

C5194 (E) Expected an asm string

C5195 (E) An asm function must be prototyped

C5196 (E) An asm function may not have an ellipsis

C5219 (F) Error while deleting file "file name"

C5220 (E) Integral value does not fit in required floating-point type

C5221 (E) Floating-point value does not fit in required floating-point type

C5222 (E) Floating-point operation result is out of range

C5223 (W) Function function name declared implicitly

C5224 (W) The format string requires additional arguments

C5225 (W) The format string ends before this argument

C5226 (W) Invalid format string conversion

C5227 (E) Macro recursion

C5228 (W) Trailing comma is nonstandard

C5229 (W) Bit field cannot contain all values of the enumerated type

C5230 (W) Nonstandard type for a bit field

C5231 (W) Declaration is not visible outside of function

C5232 (W) Old-fashioned typedef of "void" ignored

C5233 (W) Left operand is not a struct or union containing this field

C5234 (W) Pointer does not point to struct or union containing this field

C5235 (E) Variable "name" was declared with a never-completed type

C5236 (W) (I) Controlling expression is constant

C5237 (I) Selector expression is constant

C5238 (E) Invalid specifier on a parameter

C5239 (E) Invalid specifier outside a class declaration

C5240 (E) Duplicate specifier in declaration

C5241 (E) A union is not allowed to have a base class

C5242 (E) Multiple access control specifiers are not allowed

C5243 (E) Class or struct definition is missing

C5244 (E) Qualified name is not a member of class "type" or its base classes

C5245 (E) A nonstatic member reference must be relative to a specific object

C5246 (E) A nonstatic data member may not be defined outside its class

C5247 (E) Entity-kind "name" has already been defined

C5248 (E) Pointer to reference is not allowed

C5249 (E) Reference to reference is not allowed

C5250 (E) Reference to void is not allowed

C5251 (E) Array of reference is not allowed

C5252 (E) Reference entity-kind "name" requires an initializer

C5253 (E) Expected a ","

C5254 (E) Type name is not allowed

C5255 (E) Type definition is not allowed

C5256 (E) Invalid redeclaration of type name "name" (declared at line "line number")

C5257 (E) Const entity-kind "name" requires an initializer

C5258 (E) "this" may only be used inside a nonstatic member function

C5259 (E) Constant value is not known

C5260 (W) Explicit type is missing ("int" assumed)

C5261 (I) Access control not specified ("name" by default)

C5262 (E) (W) Not a class or struct name

C5263 (E) Duplicate base class name

C5264 (E) Invalid base class

C5265 (E) Entity-kind "name" is inaccessible

C5266 (E) "name" is ambiguous

C5268 (E) Declaration may not appear after executable statement in block

C5269 (E) Conversion to inaccessible base class "type" is not allowed

C5274 (E) Improperly terminated macro invocation

C5276 (E) Name followed by "::" must be a class or namespace name

C5277 (E) Invalid friend declaration

C5278 (E) A constructor or destructor may not return a value

C5279 (E) Invalid destructor declaration

C5280 (E)(W) Declaration of a member with the same name as its class

C5281 (E) Global-scope qualifier (leading "::") is not allowed

C5282 (E) The global scope has no "name"

C5283 (E) Qualified name is not allowed

C5284 (E) (W) NULL reference is not allowed

C5285 (E) Initialization with "{...}" is not allowed for object of type "type"

C5286 (E) Base class "type" is ambiguous

C5287 (E) Derived class "type" contains more than one instance of class "type"

C5288 (E) Cannot convert pointer to base class "type1" to pointer to derived class "type2" -- base class is virtual

C5289 (E) No instance of constructor "name" matches the argument list

C5290 (E) Copy constructor for class "type" is ambiguous

C5291 (E) No default constructor exists for class "type"

C5292 (E) "name" is not a nonstatic data member or base class of class "type"

C5293 (E) Indirect nonvirtual base class is not allowed

C5294 (E) Invalid union member -- class "type" has a disallowed member function

C5296 (E) (W) Invalid use of non-lvalue array

C5297 (E) Expected an operator

C5298 (E) Inherited member is not allowed

C5299 (E) Cannot determine which instance of entity-kind "name" is intended

C5300 (E) (W) A pointer to a bound function may only be used to call the function

C5301 (E) Typedef name has already been declared (with same type)

C5302 (E) Entity-kind "name" has already been defined

C5304 (E) No instance of entity-kind "name" matches the argument list

C5305 (E) Type definition is not allowed in function return type declaration

C5306 (E) Default argument not at end of parameter list

C5307 (E) Redefinition of default argument

C5308 (E) More than one instance of "name" matches the argument list:

C5309 (E) More than one instance of constructor "name" matches the argument list:

C5310 (E) Default argument of type "type1" is incompatible with parameter of type "type2"

C5311 (E) Cannot overload functions distinguished by return type alone

C5312 (E) No suitable user-defined conversion from "type1" to "type2" exists

C5313 (E) Type qualifier is not allowed on this function

C5314 (E) Only nonstatic member functions may be virtual

C5315 (E) The object has cv-qualifiers that are not compatible with the member function

C5316 (E) Program too large to compile (too many virtual functions)

C5317 (E) Return type is not identical to nor covariant with return type "type" of overridden virtual function entity-kind "name"

C5318 (E) Override of virtual entity-kind "name" is ambiguous

C5319 (E) Pure specifier ("= 0") allowed only on virtual functions

C5320 (E) Badly-formed pure specifier (only "= 0" is allowed)

C5321 (E) Data member initializer is not allowed

C5322 (E) Object of abstract class type "type" is not allowed:

C5323 (E) Function returning abstract class "type" is not allowed:

C5324 (I) Duplicate friend declaration

C5325 (E) Inline specifier allowed on function declarations only

C5326 (E) (W) "inline" is not allowed

C5327 (E) Invalid storage class for an inline function

C5328 (E) Invalid storage class for a class member

C5329 (E) Local class member entity-kind "name" requires a definition

C5330 (E) Entity-kind "name" is inaccessible

C5332 (E) Class "type" has no copy constructor to copy a const object

C5333 (E) Defining an implicitly declared member function is not allowed

C5334 (E) Class "type" has no suitable copy constructor

C5335 (E) (W) Linkage specification is not allowed

C5336 (E) Unknown external linkage specification

C5337 (E) Linkage specification is incompatible with previous "name" (declared at line "line number")

C5338 (E) More than one instance of overloaded function "name" has "C" linkage

C5339 (E) Class "type" has more than one default constructor

C5340 (E) Value copied to temporary, reference to temporary used

C5341 (E) "operator" must be a member function

C5342 (E) Operator may not be a static member function

C5343 (E) No arguments allowed on user-defined conversion

C5344 (E) Too many parameters for this operator function

C5345 (E) Too few parameters for this operator function

C5346 (E) Nonmember operator requires a parameter with class type

C5347 (E) Default argument is not allowed

C5348 (E) More than one user-defined conversion from "type1" to "type2" applies:

C5349 (E) No operator "operator" matches these operands

C5350 (E) More than one operator "operator" matches these operands:

C5351 (E) First parameter of allocation function must be of type "size_t"

C5352 (E) Allocation function requires "void *" return type

C5353 (E) Deallocation function requires "void" return type

C5354 (E) First parameter of deallocation function must be of type "void *"

C5356 (E) Type must be an object type

C5357 (E) Base class "type" has already been initialized

C5359 (E) Entity-kind "name" has already been initialized

C5360 (E) Name of member or base class is missing

C5363 (E) Invalid anonymous union -- nonpublic member is not allowed

C5364 (E) Invalid anonymous union -- member function is not allowed

C5365 (E) Anonymous union at global or namespace scope must be declared static

C5366 (E) Entity-kind "name" provides no initializer for:

C5367 (E) Implicitly generated constructor for class "type" cannot initialize:

C5368 (W) Entity-kind "name" defines no constructor to initialize the following:

C5369 (E) Entity-kind "name" has an uninitialized const or reference member

C5370 (W) Entity-kind "name" has an uninitialized const field

C5371 (E) Class "type" has no assignment operator to copy a const object

C5372 (E) Class "type" has no suitable assignment operator

C5373 (E) Ambiguous assignment operator for class "type"

C5375 (E) Declaration requires a typedef name

C5377 (W) "virtual" is not allowed

C5378 (E) "static" is not allowed

C5380 (E) Expression must have pointer-to-member type

C5381 (I) Extra ";" ignored

C5382 (W) In-class initializer for nonstatic member is nonstandard

C5384 (E) No instance of overloaded "name" matches the argument list

C5386 (E) No instance of entity-kind "name" matches the required type

C5388 (E) "operator->" for class "type1" returns invalid type "type2"

C5389 (E) A cast to abstract class "type" is not allowed:

C5390 (E) Function "main" may not be called or have its address taken

C5391 (E) A new-initializer may not be specified for an array

C5392 (E) Member function "name" may not be redeclared outside its class

C5393 (E) Pointer to incomplete class type is not allowed

C5394 (E) Reference to local variable of enclosing function is not allowed

C5397 (E) Implicitly generated assignment operator cannot copy:

C5398 (W) Cast to array type is nonstandard (treated as cast to "type")

C5399 (I) Entity-kind "name" has an operator newxxxx() but no default operator deletexxxx()

C5400 (I) Entity-kind "name" has a default operator deletexxxx() but no operator newxxxx()

C5401 (E) Destructor for base class "type" is not virtual

C5403 (E) Invalid redeclaration of member "function name"

C5404 (E) Function "main" may not be declared inline

C5405 (E) Member function with the same name as its class must be a constructor

C5407 (E) A destructor may not have parameters

C5408 (E) Copy constructor for class "type1" may not have a parameter of type "type2"

C5409 (E) Entity-kind "name" returns incomplete type "type"

C5410 (E) Protected entity-kind "name" is not accessible through a "type" pointer or object

C5411 (E) A parameter is not allowed

C5412 (E) An "asm" declaration is not allowed here

C5413 (E) No suitable conversion function from "type1" to "type2" exists

C5414 (W) Delete of pointer to incomplete class

C5415 (E) No suitable constructor exists to convert from "type1" to "type2"

C5416 (E) More than one constructor applies to convert from "type1" to "type2":

C5417 (E) More than one conversion function from "type1" to "type2" applies:

C5418 (E) More than one conversion function from "type" to a built-in type applies:

C5424 (E) A constructor or destructor may not have its address taken

C5427 (E) Qualified name is not allowed in member declaration

C5429 (E) The size of an array in "new" must be non-negative

C5430 (W) Returning reference to local temporary

C5432 (E) "enum" declaration is not allowed

C5433 (E) Qualifiers dropped in binding reference of type "type1" to initializer of type "type2"

C5434 (E) A reference of type "type1" (not const-qualified) cannot be initialized with a value of type "type2"

C5435 (E) A pointer to function may not be deleted

C5436 (E) Conversion function must be a nonstatic member function

C5437 (E) Template declaration is not allowed here

C5438 (E) Expected a "<"

C5439 (E) Expected a ">"

C5440 (E) Template parameter declaration is missing

C5441 (E) Argument list for entity-kind "name" is missing

C5442 (E) Too few arguments for entity-kind "name"

C5443 (E) Too many arguments for entity-kind "name"

C5445 (E) Entity-kind "name1" is not used in declaring the parameter types of entity-kind "name2"

C5449 (E) More than one instance of entity-kind "name" matches the required type

C5450 (E) The type "long long" is nonstandard

C5451 (E) Omission of "class" is nonstandard

C5452 (E) Return type may not be specified on a conversion function

C5456 (E) Excessive recursion at instantiation of entity-kind "name"

C5457 (E) "name" is not a function or static data member

C5458 (E) Argument of type "type1" is incompatible with template parameter of type "type2"

C5459 (E) Initialization requiring a temporary or conversion is not allowed

C5460 (W) Declaration of "variable name" hides function parameter

C5461 (E) Initial value of reference to non-const must be an lvalue

C5463 (E) "template" is not allowed

C5464 (E) "type" is not a class template

C5466 (E) "main" is not a valid name for a function template

C5467 (E) Invalid reference to entity-kind "name" (union/nonunion mismatch)

C5468 (E) A template argument may not reference a local type

C5469 (E) Tag kind of "name1" is incompatible with declaration of entity-kind "name2" (declared at line "line number")

C5470 (E) The global scope has no tag named "name"

C5471 (E) Entity-kind "name1" has no tag member named "name2"

C5473 (E) Entity-kind "name" may be used only in pointer-to-member declaration

C5475 (E) A template argument may not reference a non-external entity

C5476 (E) Name followed by "::~" must be a class name or a type name

C5477 (E) Destructor name does not match name of class "type"

C5478 (E) Type used as destructor name does not match type "type"

C5479 (I) Entity-kind "name" redeclared "inline" after being called

C5481 (E) Invalid storage class for a template declaration

C5484 (E) Invalid explicit instantiation declaration

C5485 (E) Entity-kind "name" is not an entity that can be instantiated

C5486 (E) Compiler generated entity-kind "name" cannot be explicitly instantiated

C5487 (E) (I) Inline entity-kind "name" cannot be explicitly instantiated

C5489 (E) Entity-kind "name" cannot be instantiated -- no template definition was supplied

C5490 (E) Entity-kind "name" cannot be instantiated -- it has been explicitly specialized

C5493 (E) No instance of entity-kind "name" matches the specified type

C5494 (E) (W) Declaring a void parameter list with a typedef is nonstandard

C5496 (E) Template parameter "name" may not be redeclared in this scope

C5497 (W) Declaration of "name" hides template parameter

C5498 (E) Template argument list must match the parameter list

C5500 (E) Extra parameter of postfix "operatorxxxx" must be of type "int"

C5501 (E) An operator name must be declared as a function

C5502 (E) Operator name is not allowed

C5503 (E) Entity-kind "name" cannot be specialized in the current scope

C5504 (E) Nonstandard form for taking the address of a member function

C5505 (E) Too few template parameters -- does not match previous declaration

C5506 (E) Too many template parameters -- does not match previous declaration

C5507 (E) Function template for operator delete(void *) is not allowed

C5508 (E) Class template and template parameter may not have the same name

C5510 (E) A template argument may not reference an unnamed type

C5511 (E) Enumerated type is not allowed

C5512 (W) Type qualifier on a reference type is not allowed

C5513 (E) (W) A value of type "type1" cannot be assigned to an entity of type "type2"

C5514 (W) Pointless comparison of unsigned integer with a negative constant

C5515 (E) Cannot convert to incomplete class "type"

C5516 (E) Const object requires an initializer

C5517 (E) Object has an uninitialized const or reference member

C5518 (E) Nonstandard preprocessing directive

C5519 (E) Entity-kind "name" may not have a template argument list

C5520 (E) (W) Initialization with "{...}" expected for aggregate object

C5521 (E) Pointer-to-member selection class types are incompatible ("type1" and "type2")

C5522 (W) Pointless friend declaration

C5523 (W) "." used in place of "::" to form a qualified name

C5525 (W) A dependent statement may not be a declaration

C5526 (E) A parameter may not have void type

C5529 (E) This operator is not allowed in a template argument expression

C5530 (E) Try block requires at least one handler

C5531 (E) Handler requires an exception declaration

C5532 (E) Handler is masked by default handler

C5533 (W) Handler is potentially masked by previous handler for type "type"

C5534 (I) Use of a local type to specify an exception

C5535 (I) Redundant type in exception specification

C5536 (E) Exception specification is incompatible with that of previous entity-kind "name" (declared at line "line number"):

C5540 (E) Support for exception handling is disabled

C5541 (W) Omission of exception specification is incompatible with previous entity-kind "name" (declared at line "line number")

C5542 (F) Could not create instantiation request file "name"

C5543 (E) Non-arithmetic operation not allowed in nontype template argument

C5544 (E) Use of a local type to declare a nonlocal variable

C5545 (E) Use of a local type to declare a function

C5546 (E) Transfer of control bypasses initialization of:

C5548 (E) Transfer of control into an exception handler

C5549 (I) Entity-kind "name" is used before its value is set

C5550 (W) Entity-kind "name" was set but never used

C5551 (E) Entity-kind "name" cannot be defined in the current scope

C5552 (W) Exception specification is not allowed

C5553 (W) External/internal linkage conflict for entity-kind "name" (declared at line "line number")

C5554 (W) Entity-kind "name" will not be called for implicit or explicit conversions

C5555 (E) Tag kind of "name" is incompatible with template parameter of type "type"

C5556 (E) Function template for operator new(size_t) is not allowed

C5558 (E) Pointer to member of type "type" is not allowed

C5559 (E) Ellipsis is not allowed in operator function parameter list

C5560 (E) "keyword" is reserved for future use as a keyword

C5563 (F) Invalid preprocessor output file

C5598 (E) A template parameter may not have void type

C5599 (E) Excessive recursive instantiation of entity-kind "name" due to instantiate-all mode

C5601 (E) A throw expression may not have void type

C5603 (E) Parameter of abstract class type "type" is not allowed:

C5604 (E) Array of abstract class "type" is not allowed:

C5605 (E) Floating-point template parameter is nonstandard

C5606 (E) This pragma must immediately precede a declaration

C5607 (E) This pragma must immediately precede a statement

C5608 (E) This pragma must immediately precede a declaration or statement

C5609 (E) This kind of pragma may not be used here

C5611 (W) Overloaded virtual function "name1" is only partially overridden in entity-kind "name2"

C5612 (E) Specific definition of inline template function must precede its first use

C5615 (E) Parameter type involves pointer to array of unknown bound

C5616 (E) Parameter type involves reference to array of unknown bound

C5617 (W) Pointer-to-member-function cast to pointer to function

C5618 (I) Struct or union declares no named members

C5619 (E) Nonstandard unnamed field

C5620 (E) Nonstandard unnamed member

C5624 (E) "name" is not a type name

C5641 (F) "name" is not a valid directory

C5642 (F) Cannot build temporary file name

C5643 (E) "restrict" is not allowed

C5644 (E) A pointer or reference to function type may not be qualified by "restrict"

C5647 (E) Conflicting calling convention modifiers

C5650 (W) Calling convention specified here is ignored

C5651 (E) A calling convention may not be followed by a nested declarator

C5652 (I) Calling convention is ignored for this type

C5654 (E) Declaration modifiers are incompatible with previous declaration

C5656 (E) Transfer of control into a try block

C5657 (W) Inline specification is incompatible with previous "name" (declared at line "line number")

C5658 (E) Closing brace of template definition not found

C5660 (E) Invalid packing alignment value

C5661 (E) Expected an integer constant

C5662 (W) Call of pure virtual function

C5663 (E) Invalid source file identifier string

C5664 (E) A class template cannot be defined in a friend declaration

C5665 (E) "asm" is not allowed

C5666 (E) "asm" must be used with a function definition

C5667 (E) "asm" function is nonstandard

C5668 (E) Ellipsis with no explicit parameters is nonstandard

C5669 (E) "&..." is nonstandard

C5670 (E) Invalid use of "&..."

C5673 (E) A reference of type "type1" cannot be initialized with a value of type "type2"

C5674 (E) Initial value of reference to const volatile must be an lvalue

C5676 (W) Using out-of-scope declaration of "symbol name"

C5678 (I) Call of entity-kind "name" (declared at line "line number") cannot be inlined

C5679 (I) Entity-kind "name" cannot be inlined

C5691 (E) (W) "symbol", required for copy that was eliminated, is inaccessible

C5692 (E) (W) "symbol", required for copy that was eliminated, is not callable because reference parameter cannot be bound to rvalue

C5693 (E) <typeinfo> must be included before typeid is used

C5694 (E) "name" cannot cast away const or other type qualifiers

C5695 (E) The type in a dynamic_cast must be a pointer or reference to a complete class type, or void *

C5696 (E) The operand of a pointer dynamic_cast must be a pointer to a complete class type

C5697 (E) The operand of a reference dynamic_cast must be an lvalue of a complete class type

C5698 (E) The operand of a runtime dynamic_cast must have a polymorphic class type

C5701 (E) An array type is not allowed here

C5702 (E) Expected an "="

C5703 (E) Expected a declarator in condition declaration

C5704 (E) "name", declared in condition, may not be redeclared in this scope

C5705 (E) Default template arguments are not allowed for function templates

C5706 (E) Expected a "," or ">"

C5707 (E) Expected a template parameter list

C5708 (W) Incrementing a bool value is deprecated

C5709 (E) bool type is not allowed

C5710 (E) Offset of base class "name1" within class "name2" is too large

C5711 (E) Expression must have bool type (or be convertible to bool)

C5717 (E) The type in a const_cast must be a pointer, reference, or pointer to member to an object type

C5718 (E) A const_cast can only adjust type qualifiers; it cannot change the underlying type

C5719 (E) mutable is not allowed

C5720 (W) Redeclaration of entity-kind "name" is not allowed to alter its access

C5722 (W) Use of alternative token "<:" appears to be unintended

C5723 (W) Use of alternative token "%:" appears to be unintended

C5724 (E) namespace definition is not allowed

C5725 (E) Name must be a namespace name

C5726 (E) Namespace alias definition is not allowed

C5727 (E) namespace-qualified name is required

C5728 (E) A namespace name is not allowed

C5730 (E) Entity-kind "name" is not a class template

C5731 (E) Array with incomplete element type is nonstandard

C5732 (E) Allocation operator may not be declared in a namespace

C5733 (E) Deallocation operator may not be declared in a namespace

C5734 (E) Entity-kind "name1" conflicts with using-declaration of entity-kind "name2"

C5735 (E) Using-declaration of entity-kind "name1" conflicts with entity-kind "name2" (declared at line "line number")

C5737 (W) Using-declaration ignored -- it refers to the current namespace

C5738 (E) A class-qualified name is required

C5741 (W) Using-declaration of entity-kind "name" ignored

C5742 (E) Entity-kind "name1" has no actual member "name2"

C5748 (W) Calling convention specified more than once

C5749 (E) A type qualifier is not allowed

C5750 (E) Entity-kind "name" (declared at line "line number") was used before its template was declared

C5751 (E) Static and nonstatic member functions with same parameter types cannot be overloaded

C5752 (E) No prior declaration of entity-kind "name"

C5753 (E) A template-id is not allowed

C5754 (E) A class-qualified name is not allowed

C5755 (E) Entity-kind "name" may not be redeclared in the current scope

C5756 (E) Qualified name is not allowed in namespace member declaration

C5757 (E) Entity-kind "name" is not a type name

C5758 (E) Explicit instantiation is not allowed in the current scope

C5759 (E) "symbol name" cannot be explicitly instantiated in the current scope

C5760 (W) "symbol" explicitly instantiated more than once

C5761 (E) Typename may only be used within a template

C5765 (E) Nonstandard character at start of object-like macro definition

C5766 (W) Exception specification for virtual entity-kind "name1" is incompatible with that of overridden entity-kind "name2"

C5767 (W) Conversion from pointer to smaller integer

C5768 (W) Exception specification for implicitly declared virtual entity-kind "name1" is incompatible with that of overridden entity-kind "name2"

C5769 (E) "symbol1", implicitly called from "symbol2", is ambiguous

C5771 (E) "explicit" is not allowed

C5772 (E) Declaration conflicts with "name" (reserved class name)

C5773 (E) Only "()" is allowed as initializer for array entity-kind "name"

C5774 (E) "virtual" is not allowed in a function template declaration

C5775 (E) Invalid anonymous union -- class member template is not allowed

C5776 (E) Template nesting depth does not match the previous declaration of entity-kind "name"

C5777 (E) This declaration cannot have multiple "template <...>" clauses

C5779 (E) "name", declared in for-loop initialization, may not be redeclared in this scope

C5780 (W) Reference is to "symbol1" -- under old for-init scoping rules it would have been "symbol2"

C5782 (E) Definition of virtual entity-kind "name" is required here

C5783 (W) Empty comment interpreted as token-pasting operator "##"

C5784 (E) A storage class is not allowed in a friend declaration

C5785 (E) Template parameter list for "name" is not allowed in this declaration

C5786 (E) entity-kind "name" is not a valid member class or function template

C5787 (E) Not a valid member class or function template declaration

C5788 (E) A template declaration containing a template parameter list may not be followed by an explicit specialization declaration

C5789 (E) Explicit specialization of entity-kind "name1" must precede the first use of entity-kind "name2"

C5790 (E) Explicit specialization is not allowed in the current scope

C5791 (E) Partial specialization of entity-kind "name" is not allowed

C5792 (E) Entity-kind "name" is not an entity that can be explicitly specialized

C5793 (E) Explicit specialization of entity-kind "name" must precede its first use

C5794 (W) Template parameter "template parameter" may not be used in an elaborated type specifier

C5795 (E) Specializing "name" requires "template<>" syntax

C5799 (E) Specializing "symbol name" without "template<>" syntax is nonstandard

C5800 (E) This declaration may not have extern "C" linkage

C5801 (E) "name" is not a class or function template name in the current scope

C5802 (W) Specifying a default argument when redeclaring an unreferenced function template is nonstandard

C5803 (E) Specifying a default argument when redeclaring an already referenced function template is not allowed

C5804 (E) Cannot convert pointer to member of base class "type1" to pointer to member of derived class "type2" -- base class is virtual

C5805 (E) Exception specification is incompatible with that of entity-kind "name" (declared at line "line number"):

C5806 (W) Omission of exception specification is incompatible with entity-kind "name" (declared at line "line number")

C5807 (E) Unexpected end of default argument expression

C5808 (E) Default-initialization of reference is not allowed

C5809 (E) Uninitialized entity-kind "name" has a const member

C5810 (E) Uninitialized base class "type" has a const member

C5811 (E) Const entity-kind "name" requires an initializer -- class "type" has no explicitly declared default constructor

C5812 (E) (W) Const object requires an initializer -- class "type" has no explicitly declared default constructor

C5815 (I) Type qualifier on return type is meaningless

C5816 (E) In a function definition a type qualifier on a "void" return type is not allowed

C5817 (E) Static data member declaration is not allowed in this class

C5818 (E) Template instantiation resulted in an invalid function declaration

C5819 (E) "..." is not allowed

C5822 (E) Invalid destructor name for type "type"

C5824 (E) Destructor reference is ambiguous -- both entity-kind "name1" and entity-kind "name2" could be used

C5825 (W) Virtual inline entity-kind "name" was never defined

C5826 (W) Entity-kind "name" was never referenced

C5827 (E) Only one member of a union may be specified in a constructor initializer list

C5828 (E) Support for "new[]" and "delete[]" is disabled

C5829 (W) "double" used for "long double" in generated C code

C5830 (W) "symbol" has no corresponding operator deletes (to be called if an exception is thrown during initialization of an allocated object)

C5831 (W) (I) Support for placement delete is disabled

C5832 (E) No appropriate operator delete is visible

C5833 (E) Pointer or reference to incomplete type is not allowed

C5834 (E) Invalid partial specialization -- entity-kind "name" is already fully specialized

C5835 (E) Incompatible exception specifications

C5836 (W) Returning reference to local variable

C5837 (W) Omission of explicit type is nonstandard ("int" assumed)

C5838 (E) More than one partial specialization matches the template argument list of entity-kind "name"

C5840 (E) A template argument list is not allowed in a declaration of a primary template

C5841 (E) Partial specializations may not have default template arguments

C5842 (E) Entity-kind "name1" is not used in template argument list of entity-kind "name2"

C5843 (E) The type of partial specialization template parameter entity-kind "name" depends on another template parameter

C5844 (E) The template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter

C5845 (E) This partial specialization would have been used to instantiate entity-kind "name"

C5846 (E) This partial specialization would have been made the instantiation of entity-kind "name" ambiguous

C5847 (E) Expression must have integral or enum type

C5848 (E) Expression must have arithmetic or enum type

C5849 (E) Expression must have arithmetic, enum, or pointer type

C5850 (E) Type of cast must be integral or enum

C5851 (E) Type of cast must be arithmetic, enum, or pointer

C5852 (E) Expression must be a pointer to a complete object type

C5854 (E) A partial specialization nontype argument must be the name of a nontype parameter or a constant

C5855 (E) (W) Return type is not identical to return type "type" of overridden virtual function entity-kind "name"

C5857 (E) A partial specialization of a class template must be declared in the namespace of which it is a member

C5858 (E) Entity-kind "name" is a pure virtual function

C5859 (E) Pure virtual entity-kind "name" has no overrider

C5861 (E) Invalid character in input line

C5862 (E) Function returns incomplete type "type"

C5863 (I) Effect of this "#pragma pack" directive is local to "symbol"

C5864 (E) "name" is not a template

C5865 (E) A friend declaration may not declare a partial specialization

C5866 (I) Exception specification ignored

C5867 (W) Declaration of "size_t" does not match the expected type "type"

C5868 (E) Space required between adjacent ">" delimiters of nested template argument lists (">>" is the right shift operator)

C5869 (E) Could not set locale to allow processing of multibyte characters

C5870 (W) Invalid multibyte character sequence

C5871 (E) Template instantiation resulted in unexpected function type of "type1" (the meaning of a name may have changed since the template declaration -- the type of the template is "type2")

C5872 (E) Ambiguous guiding declaration -- more than one function template no matches type "type"

C5873 (E) Non-integral operation not allowed in nontype template argument

C5875 (E) Embedded C++ does not support templates

C5876 (E) Embedded C++ does not support exception handling

C5877 (E) Embedded C++ does not support namespaces

C5878 (E) Embedded C++ does not support run-time type information

C5879 (E) Embedded C++ does not support the new cast syntax

C5880 (E) Embedded C++ does not support using-declarations

C5881 (E) Embedded C++ does not support "mutable"

C5882 (E) Embedded C++ does not support multiple or virtual inheritance

C5885 (E) "type1" cannot be used to designate constructor for "type2"

C5886 (E) Invalid suffix on integral constant

C5890 (E) Variable length array with unspecified bound is not allowed

C5891 (E) An explicit template argument list is not allowed on this declaration

C5892 (E) An entity with linkage cannot have a type involving a variable length array

C5893 (E) A variable length array cannot have static storage duration

C5894 (E) Entity-kind "name" is not a template

C5896 (E) Expected a template argument

C5898 (E) Nonmember operator requires a parameter with class or enum type

C5900 (E) Using-declaration of entity-kind "name" is not allowed

C5901 (E) Qualifier of destructor name "type1" does not match type "type2"

C5902 (W) Type qualifier ignored

C5907 (E) Option "nonstd_qualifier_deduction" can be used only when compiling C++

C5912 (W) Ambiguous class member reference – "symbol1" used in preference to "symbol2"

C5915 (E) A segment name has already been specified

C5916 (E) Cannot convert pointer to member of derived class "type1" to pointer to member of base class "type2" -- base class is virtual

C5919 (F) Invalid output file: "name"

C5920 (F) Cannot open output file: "name"

C5925 (W) Type qualifiers on function types are ignored

C5926 (F) Cannot open definition list file: "name"

C5928 (E) Incorrect use of va_start

C5929 (E) Incorrect use of va_arg

C5930 (E) Incorrect use of va_end

C5934 (E) A member with reference type is not allowed in a union

C5935 (E) "typedef" may not be specified here

C5936 (W) Redeclaration of entity-kind "name" alters its access

C5937 (E) A class or namespace qualified name is required

C5938 (E) Return type "int" omitted in declaration of function "main"

C5939 (E) pointer-to-member representation "symbol1" is too restrictive for "symbol2"

C5940 (W) Missing return statement at end of non-void entity-kind "name"

C5941 (W) Duplicate using-declaration of "name" ignored

C5942 (W) enum bit-fields are always unsigned, but enum "name" includes negative enumerator

C5946 (E) Name following "template" must be a member template

C5947 (E) Name following "template" must have a template argument list

C5948 (E) (W) Nonstandard local-class friend declaration -- no prior declaration in the enclosing scope

C5949 (I) Specifying a default argument on this declaration is nonstandard

C5951 (E) (W) Return type of function "main" must be "int"

C5952 (E) A template parameter may not have class type

C5953 (E) A default template argument cannot be specified on the declaration of a member of a class template

C5954 (E) A return statement is not allowed in a handler of a function try block of a constructor

C5955 (E) Ordinary and extended designators cannot be combined in an initializer designation

C5956 (E) The second subscript must not be smaller than the first

C5959 (W) Declared size for bit field is larger than the size of the bit field type; truncated to "bit count" bits

C5960 (E) Type used as constructor name does not match type "type"

C5961 (W) Use of a type with no linkage to declare a variable with linkage

C5962 (W) Use of a type with no linkage to declare a function

C5963 (E) Return type may not be specified on a constructor

C5964 (E) Return type may not be specified on a destructor

C5965 (E) Incorrectly formed universal character name

C5966 (E) Universal character name specifies an invalid character

C5967 (E) A universal character name cannot designate a character in the basic character set

C5968 (E) This universal character is not allowed in an identifier

C5969 (E) The identifier __VA_ARGS__ can only appear in the replacement lists of variadic macros

C5970 (W) The qualifier on this friend declaration is ignored

C5971 (E) Array range designators cannot be applied to dynamic initializers

C5972 (E) Property name cannot appear here

C5973 (W) "inline" used as a function qualifier is ignored

C5975 (E) A variable-length array type is not allowed

C5976 (E) A compound literal is not allowed in an integral constant expression

C5977 (E) A compound literal of type "type" is not allowed

C5978 (E) A template friend declaration cannot be declared in a local class

C5979 (E) Ambiguous "?" operation: second operand of type "type1" can be converted to third operand type "type2", and vice versa

C5980 (E) Call of an object of a class type without appropriate operator() or conversion functions to pointer-to-function type

C5982 (E) There is more than one way an object of type "type" can be called for the argument list

C5983 (E) typedef name has already been declared (with similar type)

C5984 (W) Operator new and operator delete cannot be given internal linkage

C5985 (E) Storage class "mutable" is not allowed for anonymous unions

C5987 (E) Abstract class type "type" is not allowed as catch type:

C5988 (E) A qualified function type cannot be used to declare a nonmember function or a static member function

C5989 (E) A qualified function type cannot be used to declare a parameter

C5990 (E) Cannot create a pointer or reference to qualified function type

C5991 (W) Extra braces are nonstandard

C5992 (E) Invalid macro definition:

C5993 (W) Subtraction of pointer types "symbol name1" and "symbol name2" is nonstandard

C5994 (E) An empty template parameter list is not allowed in a template parameter declaration

C5995 (E) Expected "class"

C5996 (E) The "class" keyword must be used when declaring a template parameter

C5997 (W) "function name1" is hidden by "function name2" -- virtual function override intended?

C5998 (E) A qualified name is not allowed for a friend declaration that is a function definition

C5999 (E) "type1" is not compatible with "type2"

C6000 (W) A storage class may not be specified here

C6001 (E) Class member designated by a using-declaration must be visible in a direct base class

C6006 (E) A template parameter cannot have the same name as one of its template parameters

C6007 (E) Recursive instantiation of default argument

C6009 (E) "instance name" is not an entity that can be defined

C6010 (E) Destructor name must be qualified

C6011 (E) Friend class name may not be introduced with "typename"

C6012 (E) A using-declaration may not name a constructor or destructor

C6013 (E) A qualified friend template declaration must refer to a specific previously declared template

C6014 (E) Invalid specifier in class template declaration

C6015 (E) Argument is incompatible with formal parameter

C6017 (E) Loop in sequence of "operator->" functions starting at class "symbol"

C6018 (E) "class name" has no member class "member name"

C6019 (E) The global scope has no class named "class name"

C6020 (E) Recursive instantiation of template default argument

C6021 (E) Access declarations and using-declarations cannot appear in unions

C6022 (E) "name" is not a class member

C6023 (E) Nonstandard member constant declaration is not allowed

C6028 (W) Invalid redeclaration of nested class

C6029 (E) Type containing an unknown-size array is not allowed

C6030 (W) A variable with static storage duration cannot be defined within an inline function

C6031 (W) An entity with internal linkage cannot be referenced within an inline function with external linkage

C6032 (E) Argument type "type" does not match this type-generic function macro

C6034 (E) Friend declaration cannot add default arguments to previous declaration

C6035 (E) "template name" cannot be declared in this scope

C6036 (E) The reserved identifier "symbol" may only be used inside a function

C6037 (E) This universal character cannot begin an identifier

C6038 (E) Expected a string literal

C6039 (E) Unrecognized STDC pragma

C6040 (E) Expected "ON", "OFF", or "DEFAULT"

C6041 (E) A STDC pragma may only appear between declarations in the global scope or before any statements or declarations in a block scope

C6042 (E) Incorrect use of va_copy

C6043 (E) "type" can only be used with floating-point types

C6044 (E) Complex type is not allowed

C6045 (E) Invalid designator kind

C6046 (W) Floating-point value cannot be represented exactly

C6047 (E) Complex floating-point operation result is out of range

C6048 (E) Conversion between real and imaginary yields zero

C6049 (E) An initializer cannot be specified for a flexible array member

C6050 (W) imaginary *= imaginary sets the left-hand operand to zero

C6051 (E) (W) Standard requires that "symbol" be given a type by a subsequent declaration ("int" assumed)

C6052 (E) A definition is required for inline "symbol"

C6053 (W) Conversion from integer to smaller pointer

C6054 (E) A floating-point type must be included in the type specifier for a _Complex or _Imaginary type

C6055 (E) Types cannot be declared in anonymous unions

C6056 (W) Returning pointer to local variable

C6057 (W) Returning pointer to local temporary

C6061 (E) Declaration of "symbol name" is incompatible with a declaration in another translation unit

C6062 (E) The other declaration is "line"

C6065 (E) A field declaration cannot have a type involving a variable length array

C6066 (E) declaration of "instance" had a different meaning during compilation of "symbol"

C6067 (E) Expected "template"

C6072 (E) (W) A declaration cannot have a label

C6075 (E) "instance name" already defined during compilation of "symbol"

C6076 (E) "symbol" already defined in another translation unit

C6081 (E) A field with the same name as its class cannot be declared in a class with a user-declared constructor

C6083 (F) Exported template file file name is corrupted

C6086 (E) the object has cv-qualifiers that are not compatible with the member "symbol"

C6087 (E) No instance of "class name" matches the argument list and object (the object has cv-qualifiers that prevent a match)

C6089 (E) There is no type with the width specified

C6105 (W) #warning directive: "string"

C6139 (E) The "template" keyword used for syntactic disambiguation may only be used within a template

C6144 (E) Storage class must be auto or register

C6145 (W) "type1" would have been promoted to "type2" when passed through the ellipsis parameter; use the latter type instead

C6146 (E) "symbol" is not a base class member

C6151 (F) Mangled name is too long

C6158 (E) void return type cannot be qualified

C6161 (E) A member template corresponding to "symbol" is declared as a template of a different kind in another translation unit

C6163 (E) va_start should only appear in a function with an ellipsis parameter

C6192 (W) Null (zero) character in input line ignored

C6193 (W) Null (zero) character in string or character constant

C6194 (W) Null (zero) character in header name

C6197 (W) The prototype declaration of "symbol" is ignored after this unprototyped redeclaration

C6201 (E) Typedef "symbol" may not be used in an elaborated type specifier

C6203 (E) Parameter "parameter name" may not be redeclared in a catch clause of function try block

C6204 (E) The initial explicit specialization of "symbol name" must be declared in the namespace containing the template

C6206 (E) "template" must be followed by an identifier

C6211 (W) Nonstandard cast to array type ignored

C6212 (E) This pragma cannot be used in a _Pragma operator (a #pragma directive must be used)

C6213 (W) Field uses tail padding of a base class

C6218 (W) Base class "class name1" uses tail padding of base class "class name2"

C6222 (W) Invalid error number

C6223 (W) Invalid error tag

C6224 (W) Expected an error number or error tag

C6227 (E) Transfer of control into a statement expression is not allowed

C6229 (E) This statement is not allowed inside of a statement expression

C6230 (E) A non-POD class definition is not allowed inside of a statement expression

C6235 (W) Nonstandard conversion between pointer to function and pointer to data

C6254 (E) Integer overflow in internal computation due to size or complexity of "type"

C6255 (E) Integer overflow in internal computation

C6273 (W) Alignment-of operator applied to incomplete type

C6280 (E) Conversion from inaccessible base class "class name" is not allowed

C6282 (E) String literals with different character kinds cannot be concatenated

C6285 (W) Nonstandard qualified name in namespace member declaration

C6290 (W) Non-POD class type passed through ellipsis

C6291 (E) A non-POD class type cannot be fetched by va_arg

C6292 (E) The 'u' or 'U' suffix must appear before the 'l' or 'L' suffix in a fixed-point literal

C6294 (W) Integer operand may cause fixed-point overflow

C6295 (E) Fixed-point constant is out of range

C6296 (W) Fixed-point value cannot be represented exactly

C6297 (W) Constant is too large for long long; given unsigned long long type (nonstandard)

C6301 (W) "symbol" declares a non-template function -- add <> to refer to a template instance

C6302 (W) Operation may cause fixed-point overflow

C6303 (E) Expression must have integral, enum, or fixed-point type

C6304 (E) Expression must have integral or fixed-point type

C6307 (W) Class member typedef may not be redeclared

C6308 (W) Taking the address of a temporary

C6310 (W) Fixed-point value implicitly converted to floating-point type

C6311 (E) Fixed-point types have no classification

C6312 (E) A template parameter may not have fixed-point type

C6313 (E) Hexadecimal floating-point constants are not allowed

C6315 (E) Floating-point value does not fit in required fixed-point type

C6316 (W) Value cannot be converted to fixed-point value exactly

C6317 (E) Fixed-point conversion resulted in a change of sign

C6318 (E) Integer value does not fit in required fixed-point type

C6319 (E) (W) Fixed-point operation result is out of range

C6320 (E) Multiple named address spaces

C6321 (E) Variable with automatic storage duration cannot be stored in a named address space

C6322 (E) Type cannot be qualified with named address space

C6323 (E) Function type cannot be qualified with named address space

C6324 (E) Field type cannot be qualified with named address space

C6325 (E) Fixed-point value does not fit in required floating-point type

C6326 (E) Fixed-point value does not fit in required integer type

C6327 (E) Value does not fit in required fixed-point type

C6335 (F) Cannot open predefined macro file: "file name"

C6336 (F) Invalid predefined macro entry at line "line count": "macro name"

C6337 (F) Invalid macro mode name "macro mode name"

C6338 (F) Incompatible redefinition of predefined macro "macro name"

C6342 (W) const_cast to enum type is nonstandard

C6344 (E) A named address space qualifier is not allowed here

C6345 (E) An empty initializer is invalid for an array with unspecified bound

C6346 (W) Function returns incomplete class type "class name"

C6348 (I) Declaration hides "variable name"

C6349 (E) A parameter cannot be allocated in a named address space

C6350 (E) Invalid suffix on fixed-point or floating-point constant

C6351 (E) A register variable cannot be allocated in a named address space

C6352 (E) Expected "SAT" or "DEFAULT"

C6353 (I) "symbol name" has no corresponding member operator delete "symbol name" (to be called if an exception is thrown during initialization of an allocated object)

C6355 (E) A function return type cannot be qualified with a named address space

C6361 (W) Negation of an unsigned fixed-point value

C6365 (E) Named-register variables cannot have void type

C6372 (E) Nonstandard qualified name in global scope declaration

C6373 (W) Implicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)

C6374 (W) Explicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)

C6375 (W) Conversion from pointer to same-sized integral type (potential portability problem)

C6380 (E) (I) Virtual "function name" was not defined (and cannot be defined elsewhere because it is a member of an unnamed namespace)

C6381 (E) (I) Carriage return character in source line outside of comment or character/string literal

C6382 (E) Expression must have fixed-point type

C6386 (W) Storage specifier ignored

C6396 (W) White space between backslash and newline in line splice ignored

C6398 (E) Invalid member for anonymous member class -- class "symbol" has a disallowed member function

C6400 (W) Positional format specifier cannot be zero

C6403 (E) A variable-length array is not allowed in a function return type

C6404 (E) Variable-length array type is not allowed in pointer to member of type "type"

C6405 (E) The result of a statement expression cannot have a type involving a variable-length array

C6420 (E) (W) Some enumerator values cannot be represented by the integral type underlying the enum type

C6421 (E) Default argument is not allowed on a friend class template declaration

C6422 (W) Multicharacter character literal (potential portability problem)

C6424 (E) Second operand of offsetof must be a field

C6425 (E) Second operand of offsetof may not be a bit field

C6426 (E) Cannot apply offsetof to a member of a virtual base

C6427 (W) offsetof applied to non-POD types is nonstandard

C6428 (E) Default arguments are not allowed on a friend declaration of a member function

C6429 (E) Default arguments are not allowed on friend declarations that are not definitions

C6430 (E) Redeclaration of "function name" previously declared as a friend with default arguments is not allowed

C6431 (E) Invalid qualifier for "symbol" (a derived class is not allowed here)

C6432 (E) Invalid qualifier for definition of class "class name"

C6439 (E) Template argument list of "symbol" must match the parameter list

C6440 (E) An incomplete class type is not allowed

C6445 (E) Invalid redefinition of "symbol name"

C6449 (E) Explicit specialization of "symbol" must precede its first use "symbol2"

C6623 (W) The destructor for "class1" has been suppressed because the destructor for "class2" is inaccessible

C6648 (W) '=' assumed following macro name "macro name" in command-line definition

C6649 (E) (W) White space is required between the macro name "macro name" and its replacement text

C6655 (E) "symbol" cannot be declared inline after its definition "definition name"

C6671 (W) __assume expression with side effects discarded

C6674 (E) __evenaccess qualifier is applied to only integer type

C6675 (E) Expected a section name string

C6676 (E) Expected a section name

C6677 (E) Invalid pragma declaration

C6678 (E) "symbol name" has already been specified by other pragma

C6679 (E) Pragma may not be specified after definition

C6680 (E) Invalid kind of pragma is specified to this symbol

C6681 (I) This pragma has no effect

C6682 (E) "symbol name" must be qualified for function type

C6683 (E) Illegal "pragma name" specifier

C6684 (E) Multiple pointer qualifiers

C6685 (E) __ptr16 must be qualified for data pointer type

C6686 (E) Invalid binary digit

C6687 (W) This pragma "name" is ignored

C6688 (E) "this" pointer of "class name" is cast implicitly to near pointer

C6689 (E) Can not specify near or far for member

C6690 (E) A member "function name" qualified with near or far is declared

C6691 (E) near or far specifier on a reference type is not allowed

C6692 (E) can not specify near or far for member function

C6693 (E) can not specify near or far for function types

C6698 (E) Incorrect PIC address usage

C6699 (E) Incorrect PID address usage

11.3 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.

Table 11.1 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, stroull, 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.	atoi, atol, atoll, 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.	strtod, fscanf, scanf, sscanf, atof
0x04B6 (EEXPN)	`Normalized exponent too large` The exponent exceeds three digits when the string literal is normalized to the IEEE standard decimal format.	strtod, fscanf, scanf, sscanf, atof
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	fscanf, scanf, sscanf

Error No.	Error Message/Explanation	Functions to Set Error Code
	(underflow).	
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

Section 12 Assembler Error Messages

12.1 Error Format and Error Levels

This section gives a list of error messages and explains details of errors in the following format.

Error number **(Error level) Error message**
Error details

There are three different error levels, corresponding to different degrees of seriousness.

Error Level	Error Type	Description
(W)	Warning	Processing is continued.
(E)	Error	Processing is interrupted.
(F)	Fatal	Processing is interrupted.

12.2 List of Messages

A1000 (W) '.ALIGN' with not 'ALIGN' specified relocatable section
Directive command **.ALIGN** is written in a section that does not have an **ALIGN** specification. 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.

A1001 (W) Destination address may be changed
The jump address can be a position that differs from an anticipated destination. 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.

A1002 (W) Floating point value is out of range
The floating-point value is out of range.
Check the floating-point value written in the source code. The value out of range is ignored.

A1003 (W) Location counter exceed
The location counter value has exceeded 0FFFFFFFFh.
Check the value of the operand in **.ORG**. Correct the source code.

A1004 (W) '.ALIGN' size is different
The specified boundary alignment value does not match the other settings.
Check the alignment value.

A1006 (W) Data in 'CODE' section align in 4byte
When **endian=big** is specified, the start address of the data area in the **CODE** section is aligned to a 4-byte boundary.

A1007 (W) Data size in 'CODE' section align in 4byte
When **endian=big** is specified, the size of the data area in the **CODE** section is adjusted to a multiple of 4.

A1009 (W) Multiple symbols
.STACK (stack value setting) is specified multiple times for a single symbol.

A1010 (W) Section attribute mismatch
The specified section attribute does not match the other settings.

A1011 (W) Use PM instruction
A privileged instruction is used.

A1012 (W) Use FPU instruction
A floating-point operation instruction is used.

A1013 (W) Use DSP instruction
A DSP function instruction is used

A1014 (W) Too many actual macro parameters
There are too many actual macro parameters.
Extra macro parameters will be ignored.

A1015 (W) Actual macro parameters are not enough
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.

A1016 (W) '.END' statement is in include file
The include file contains an **.END** statement.
.END cannot be written in include files. Delete this statement. The software will ignore **.END** as it executes.

A2000 (E) No space after mnemonic or directive
The mnemonic or assemble directive is not followed by a space character.
Enter a space character between the instruction and operand.

A2001 (E) ',' is missing
',' is not entered. Insert a comma to separate between operands.

A2002 (E) Characters exist in expression
Extra characters are written in an instruction or expression.
Check the rules to be followed when writing an expression.

A2003 (E) Size specifier is missing
No size specifier is entered.
Write a size specifier.

A2004 (E) Invalid operand(s) exist in instruction
The instruction contains an invalid operand.
Check the syntax for this instruction and rewrite it correctly.

A2005 (E) Operand type is not appropriate
The operand type is incorrect.
Check the syntax for this operand and rewrite it correctly.

A2006 (E) Size specifier is not appropriate
The size specifier is written incorrectly.
Rewrite the size specifier correctly.

A2007 (E) Operand label is not in the same section
The branch destination is not in the same section.
Execution can branch only to a destination within the same section. Correct the mnemonic.

A2008 (E) Illegal displacement value
An illegal displacement value is specified.
Specify a multiple of 2 when the size specifier is **W**. Specify a multiple of 4 when the size specifier is **L**.

A2009 (E) FPU instruction or FPSW is used
A floating-point operation (FPU) instruction or FPSW is used. Check the CPU type.

A2022 (E) Symbol name is missing
Symbol is not entered.
Write a symbol name.

A2023 (E) Illegal directive command is used
An illegal instruction is entered.
Rewrite the instruction correctly.

A2024 (E) No ';' at the top of comment
';' is not entered at the beginning of a comment.
Enter a semicolon at the beginning of each comment. Check whether the mnemonic or operand is written correctly.

A2026 (E) 'CODE' section in big endian is not appropriate
The value specified for the start address of the absolute-addressing **CODE** section is not a multiple of 4 while **endian=big** is specified.
Specify a multiple of 4 for the start address.

A2027 (E) Illegal character code
An illegal character code is specified.

A2028 (E) Unrecognized character escape sequence
An unrecognizable escape sequence is specified.

A2029 (E) Invalid description in #pragma inline_asm function
Invalid assembly-language code was used in 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.

A2040 (E) Include nesting over
Include is nested too many levels.
Rewrite include so that it is nested within 30 levels.

A2041 (E) Can't open include file 'XXXX'
The include file cannot be opened.
Check the include file name. Check the directory where the include file is stored.

A2042 (E) Including the include file in itself
An attempt is made to include the include file in itself.
Check the include file name and rewrite correctly.

A2049 (E) Invalid reserved word exist in operand
The operand contains a reserved word.
Reserved words cannot be written in an operand. Rewrite the operand correctly.

A2050 (E) Operand value is not defined
An undefined operand value is entered.
Write a valid value for operands.

A2051 (E) '{' is missing
'{' is not specified.

A2052 (E) Addressing mode specifier is not appropriate
The addressing mode specifier is written incorrectly.
Make sure that the addressing mode is written correctly.

A2053 (E) Reserved word is missing
No reserved word is entered.

A2054 (E) ']' is missing
']' is not entered.
Write the right bracket ']' corresponding to the '['.

A2055 (E) Right quote is missing
A right quote is not entered.
Enter the right quote.

A2056 (E) The value is not constant
The value is indeterminate when assembled.
Write an expression, symbol name, or label name that will have a determinate value when assembled.

A2057 (E) Quote is missing
Quotes for a character string are not entered.
Enclose a character string with quotes as you write it.

A2058 (E) Illegal operand is used
The operand is incorrect.
Check the syntax for this operand and rewrite it correctly.

A2059 (E) Operand number is not enough
The number of operands is insufficient.
Check the syntax for these operands and rewrite them correctly.

A2060 (E) Too many macro nesting
The macro is nested too many levels.
Make sure that the macro is nested no more than 65,535 levels. Check the syntax for this source statement and rewrite it correctly.

A2061 (E) Too many macro local label definition
Too many macro local labels are defined.
Make sure that the number of macro local labels defined in one file are 65,535 or less.

A2062 (E) '.MACRO' is missing for '.ENDM'
.MACRO for **.ENDM** is not found.
Check the position where **.ENDM** is written.

A2063 (E) '.MREPEAT' is missing for '.ENDR'
.MREPEAT for **.ENDR** is not found.
Check the position where **.ENDR** is written.

A2064 (E) '.MACRO' or '.MREPEAT' is missing for '.EXITM'
.MACRO or **.MREPEAT** for **.EXITM** is not found.
Check the position where **.EXITM** is written.

A2065 (E) No macro name
No macro name is entered.
Write a macro name for each macro definition.

A2066 (E) Too many formal parameter
There are too many formal parameters defined for the macro.
Make sure that the number of formal parameters defined for the macro is 80 or less.

A2067 (E) Illegal macro parameter
The macro parameter contains some incorrect description.
Check the written contents of the macro parameter.

A2068 (E) Source line is too long
The source line is excessively long.
Check the contents written in the source line and correct it as necessary.

A2069 (E) '.MACRO' is missing for '.LOCAL'
.MACRO for **.LOCAL** is not found.
Check the position where **.LOCAL** is written. **.LOCAL** can only be written in a macro block.

A2070 (E) No '.ENDM' statement
.ENDM is not entered.
Check the position where **.ENDM** is written. Write **.ENDM** as necessary.

A2071 (E) No '.ENDR' statement
.ENDR is not entered.
Check the position where **.ENDR** is written. Write **.ENDR** as necessary.

A2072 (E) ')' is missing
')' is not entered.
Write the right parenthesis ')' corresponding to the '('.

A2073 (E) Operand expression is not completed
The operand description is not complete.
Check the syntax for this operand and rewrite it correctly.

A2074 (E) Syntax error in expression
The expression is written incorrectly.
Check the syntax for this expression and rewrite it correctly.

A2075 (E) String value exist in expression
A character string is entered in the expression.
Rewrite the expression correctly.

A2076 (E) Division by zero
A divide by 0 operation is attempted.
Rewrite the expression correctly.

A2077 (E) No '.END' statement
.END is not entered.
Be sure to enter **.END** in the last line of the source program.

A2078 (E) The specified address overlaps at 'address'
Something has already been allocated to 'address'.
Check the specifications for **.ORG** and **.OFFSET**.

If the source code was C or C++, 'address' has been specified for two or more variables.
Check the variable you are attempting to allocate to 'address'.

A2080 (E) '.IF' is missing for '.ELSE'
.IF for **.ELSE** is not found.
Check the position where **.ELSE** is written.

A2081 (E) '.IF' is missing for '.ELIF'
.IF for **.ELIF** is not found.
Check the position where **.ELIF** is written.

A2082 (E) '.IF' is missing for '.ENDIF'
.IF for **.ENDIF** is not found.
Check the position where **.ENDIF** is written.

A2083 (E) Too many nesting level of condition assemble
Condition assembling is nested too many levels.
Check the syntax for this condition assemble statement and rewrite it correctly.

A2084 (E) No '.ENDIF' statement
No corresponding **ENDIF** is found for the **IF** statement in the source file.
Check the source description.

A2088 (E) Can't open '.ASSERT' message file 'XXXX'
The **.ASSERT** output file cannot be opened.
Check the file name.

A2089 (E) Can't write '.ASSERT' message file 'XXXX'
Data cannot be written to the **.ASSERT** output file.
Check the permission of the file.

A2090 (E) Too many temporary label
There are too many temporary labels.
Replace the temporary labels with label names.

A2091 (E) Temporary label is undefined
The temporary label is not defined yet.
Define the temporary label.

A2100 (E) Value is out of range
The value is out of range.
Write a value that matches the register bit length.

A2111 (E) Symbol is undefined
The symbol is not defined yet.
Undefined symbols cannot be used. Forward referenced symbol names cannot be entered.
Check the symbol name.

A2112 (E) Symbol is missing
Symbol is not entered.
Write a symbol name.

A2113 (E) Symbol definition is not appropriate
The symbol is defined incorrectly.
Check the method for defining this symbol and rewrite it correctly.

A2114 (E) Symbol has already defined as another type
The symbol has already been defined in a different directive with the same name.
Change the symbol name.

A2115 (E) Symbol has already defined as the same type
The symbol has already been defined.
Change the symbol name.

A2116 (E) Symbol is multiple defined
The symbol is defined twice or more. The macro name and some other name are duplicates.
Change the symbol name.

A2117 (E) Invalid label definition
An invalid label is entered.
Rewrite the label definition.

A2118 (E) Invalid symbol definition
An invalid symbol is entered.
Rewrite the symbol definition.

A2119 (E) Reserved word is used as label or symbol
Reserved word is used as a label or symbol.
Rewrite the label or symbol name correctly.

A2130 (E) No '.SECTION' statement
.SECTION is not entered.
Always make sure that the source program contains at least one **.SECTION**.

A2131 (E) Section type is not appropriate
An instruction or a directive used in a section does not match the section type.

A2132 (E) Section has already determined as attribute
The attribute of this section has already been defined as relative. Directive command **.ORG** cannot be written here.
Check the attribute of the section.

A2133 (E) Section attribute is not defined
Section attribute is not defined. Directive command **.ALIGN** cannot be written in this section.
Make sure that directive **.ALIGN** is written in an absolute attribute section or a relative attribute section where **ALIGN** is specified.

A2134 (E) Section name is missing
No section name is entered.
Write a section name in the operand.

A2135 (E) 'ALIGN' is multiple specified in '.SECTION'
Two or more **ALIGN**'s are specified in the **.SECTION** definition line.
Delete extra **ALIGN** specifications.

A2136 (E) Section type is multiple specified
Section type is specified two or more times in the section definition line.
Only one section type **CODE**, **DATA**, or **ROMDATA** can be specified in a section definition line.

A2137 (E) Too many operand
There are extra operands.
Check the syntax for these operands and rewrite them correctly.

A3000 (F) Can't create file 'filename'
The **filename** file cannot be generated.
Check the directory capacity.

A3001 (F) Can't open file 'filename'
The **filename** file cannot be opened.
Check the file name.

A3002 (F) Can't write file 'filename'
The **filename** file cannot be written to.
Check the permission of the file.

A3003 (F) Can't read file 'filename'
The **filename** file cannot be read.
Check the permission of the file.

A3004 (F) Can't create Temporary file
Temporary file cannot be generated.
Specify a directory in environment variable **TMP_RX** so that a temporary file will be created in some place other than the current directory.

A3005 (F) Can't open Temporary file
The temporary file cannot be opened.
Check the directory specified in **TMP_RX**.

A3006 (F) Can't read Temporary file
The temporary file cannot be read.
Check the directory specified in **TMP_RX**.

A3007 (F) Can't write Temporary file
The temporary file cannot be written to.
Check the directory specified in **TMP_RX**.

A3008 (F) Illegal file name 'filename'
The file name is illegal.
Specify a file name that conforms to file name description rules.

A3100 (F) Command line is too long
The command line has too many characters.
Re-input the command.

A3101 (F) Invalid option 'xx' is used
An invalid command option xx is used.
The specified option is nonexistent. Re-input the command correctly.

A3102 (F) Ignore option 'xx'
An invalid option is specified.

A3103 (F) Option 'xx' is not appropriate
Command option **xx** is written incorrectly.
Specify the command option correctly again.

A3104 (F) No input files specified
No input file is specified.
Specify an input file.

A3105 (F) Source files number exceed 80
The number of source files exceeds 80.
Execute assembling separately in two or more operations.

A3106 (F) Lacking cpu specification
No CPU type is specified.
Specify the CPU type by the **cpu** option or environment variable **CPU_RX**.

A3110 (F) Multiple register base/fint_register
A single register is specified by the **base** and **fint_register** options.

A3111 (F) Multiple register base/pid
A single register is specified by the **base** and **pid** options.

A3112 (F) Multiple register base/nouse_pid_register
A single register is specified by the **base** and **nouse_pid_register** options.

A3200 (F) Error occurred in executing 'xxx'
An error occurred when executing **xxx**.
Rerun **asrx**.

A3201 (F) Not enough memory
Memory is insufficient.
Divide the file and re-run. Or increase the memory capacity.

A3202 (F) Can't find work dir
The work directory is not found.
Make sure that the setting of environment variable **TMP_RX** is correct.

A4000-A4999 (-) Internal error

An internal error occurred during assembly. Report the error occurrence to your local Renesas sales office.

Section 13 Error Messages for the Optimizing Linkage Editor

13.1 Error Format and Error Levels

This section gives a list of error messages and explains details of errors in the following format.

Error number **(Error level) Error message**
Error details

There are five different error levels, corresponding to different degrees of seriousness.

Error Number	Error Level	Error Type	Description
L0000–L0999 P0000–P0999	(I)	Information	Processing is continued.
L1000–L1999 P1000–P1999	(W)	Warning	Processing is continued.
L2000–L2999 P2000–P2999	(E)	Error	Option analysis processing is continued; processing is interrupted.
L3000–L3999 P3000–P3999	(F)	Fatal	Processing is interrupted.
L4000– P4000–	(–)	Internal	Processing is interrupted.

Error numbers beginning with an **L** are optimizing linkage editor output messages.

Error numbers beginning with a **P** are prelinker output messages. Output of errors with numbers beginning with a **P** cannot be controlled using the **nomessage** or **change_message** options.

13.2 Return Values for Errors

When terminating execution, the optimizing linkage editor returns a numeric value to the OS indicating the processing result as shown below.

Return Value	Description
0	Processing was completed successfully, or processing was terminated after an information message or a warning message was output.
1	An error, a fatal error, or an internal error occurred and processing was forcibly terminated.

13.3 List of Messages

L0001 (I) Section "section" created by optimization "optimization"
The section named **section** was created as a result of the optimization.

L0002 (I) Symbol "symbol" created by optimization "optimization"
The symbol named **symbol** was created as a result of the optimization.

L0003 (I) "file"-"symbol" moved to "section" by optimization
As a result of **variable_access** optimization, the symbol named **symbol** in **file** was moved.

L0004 (I) "file"-"symbol" deleted by optimization
As a result of **symbol_delete** optimization, the symbol named **symbol** in **file** was deleted.

L0005 (I) The offset value from the symbol location has been changed by optimization : "file"-"section"-"symbol ± offset"
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**.

L0100 (I) No inter-module optimization information in "file"
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. Note however that the **goptimize** option is not available in **asmsh**.

L0101 (I) No stack information in "file"
No stack information was found in **file**. **file** may be an assembler output file or a **SYSROF**-> **ELF** converted file. The contents of the file will not be in the stack information file output by the optimizing linkage editor.

L0102 (I) Stack size "size" specified to the undefined symbol "symbol" in "file"
Stack size **size** is specified for the undefined symbol named **symbol** in **file**.

L0103 (I) Multiple stack sizes specified to the symbol "symbol"
Multiple stack sizes are specified for the symbol named **symbol**.

L0300 (I) Mode type "mode type 1" in "file" differ from "mode type 2"
A file with a different mode type was input.

L0400 (I) Unused symbol "file"–"symbol"
The symbol named **symbol** in **file** is not used.

L0500 (I) Generated CRC code at "address"
Generated CRC code at **address**.

L0510 (I) Section "section" was moved other area specified in option "cpu=<attribute>"
section without dividing is allocated according to **cpu=<attribute>**.

L0511 (I) Sections "section name","new section name" are Non-contiguous
section was divided and the newly created section is **new section name**.

L1000 (W) Option "option" ignored
The option named **option** is invalid, and is ignored.

L1001 (W) Option "option 1" is ineffective without option "option 2"
option 1 needs specifying **option 2**. **option 1** is ignored.

L1002 (W) Option "option 1" cannot be combined with option "option 2"
option 1 and **option 2** cannot be specified simultaneously. **option 1** is ignored.

L1003 (W) Divided output file cannot be combined with option "option"
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.

L1004 (W) Fatal level message cannot be changed to other level : "number"
The level of a fatal error type message cannot be changed. The specification of **number** is ignored. Only errors at the information/warning/error level can be changed with the **change_message** option.

L1005 (W) Subcommand file terminated with end option instead of exit option
There is no processing specification following the **end** option. Processing is done with the **exit** option assumed.

L1006 (W) Options following exit option ignored
All options following the **exit** option is ignored.

L1007 (W) Duplicate option : "option"
Duplicate specifications of **option** were found. Only the last specification is effective.

L1008 (W) Option "option" is effective only in cpu type "CPU type"
option is effective only in **CPU type**. **option** is ignored.

L1010 (W) Duplicate file specified in option "option" : "file"
option was used to specify the same file twice. The second specification is ignored.

L1011 (W) Duplicate module specified in option "option" : "module"
option was used to specify the same module twice. The second specification is ignored.

L1012 (W) Duplicate symbol/section specified in option "option" : "name"
option was used to specify the same symbol name or section name twice. The second specification is ignored.

L1013 (W) Duplicate number specified in option "option" : "number"
option was used to specify the same error number. Only the last specification is effective.

L1100 (W) Cannot find "name" specified in option "option"
The symbol name or section name specified in **option** cannot be found. The **name** specification is ignored.

L1101 (W) "name" in rename option conflicts between symbol and section
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.

L1102 (W) Symbol "symbol" redefined in option "option"
The symbol specified by **option** has already been defined. Processing is continued without any change.

L1103 (W) Invalid address value specified in option "option" : "address"
address specified by **option** is invalid. The **address** specification is ignored.

L1104 (W) Invalid section specified in option "option" : "section"
An invalid section was specified in **option**. Confirm the following:
(1) The **-output** option does not accept a section that has no initial value.
(2) The **-jump_entries_for_pic** option accepts only a program section.

L1110 (W) Entry symbol "symbol" in entry option conflicts
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.

L1120 (W) Section address is not assigned to "section"
section has no addresses specified for it. **section** will be located at the rearmost address. Specify the address of the section using the optimizing linkage editor option **-start**.

L1121 (W) Address cannot be assigned to absolute section "section" in start option
section is an absolute address section. An address assigned to an absolute address section is ignored.

L1122 (W) Section address in start option is incompatible with alignment : "section"
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.

L1130 (W) Section attribute mismatch in rom option : "section 1, section 2"
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**.

L1140 (W) Load address overflowed out of record-type in option "option"
A **record** type smaller than the address value was specified. The range exceeding the specified **record** type has been output as different **record** type.

L1141 (W) Cannot fill unused area from "address" with the specified value
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.

L1150 (W) Sections in "option" option have no symbol
The section specified in **option** does not have an externally defined symbol.

L1160 (W) Undefined external symbol "symbol"
An undefined external symbol **symbol** was referenced.

L1170 (W) Specified SBR addresses conflict
Different **SBR** addresses have been specified. Processing is done with **SBR=USER** assumed.

L1171 (W) Least significant byte in SBR="constant" ignored
The least significant 8 bits in address **constant** specified by the **SBR** option are ignored.

L1180 (W) Directive command "directive" is duplicated in "file"
directive was specified in multiple source files.
directive cannot be written more than once across files.

L1181 (W) Fail to write "type of output code"
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 ID code -> **ID Code**
→**L1181 Fail to write "ID Code"**
When failed to write PROTECT/OFSREG code -> **Protect Code** or **OFSREG Code**
→**L1181 Fail to write "Protect Code" or "OFSREG Code"**
When failed to write CRC code -> **CRC Code**
→**L1181 Fail to write "CRC Code"**

L1182 (W) Cannot generate vector table section "section"
The input file contains vector table **section**. The linkage editor does not create the **section** automatically.

L1183 (W) Interrupt number "vector number" of "section" is defined in input file
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.

L1190 (W) Section "section" was moved other area specified in option "cpu=<memory attribute>"
The object size was modified through optimization of access to external variables.
Accordingly, the **section** in the area specified by the next **cpu** specification was moved.

L1191 (W) Area of "FIX" is within the range of the area specified by "cpu=<memory type>" :"<start>-<end>"
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.

L1192 (W) Bss Section "section name" is not initialized
section name, which is a data section without an initial value, cannot be initialized by the initial setup program. Check the address range specified with **–cpu** and the sizes of pointer variables.

L1193 (W) Section "section name" specified in option "option" is ignored
option specified for the section newly created due to **-cpu=stride** is invalid. Do not specify **option** for the newly created section.

L1194 (W) Section "option" in relocation "file"-"section"-"offset" is changed.
The relocation **section file offset** now refers to a location in the new section created with the division of **section**. To prevent division, declare the **contiguous_section** option for **section**.

L1200 (W) Backed up file "file 1" into "file 2"
Input file **file 1** was overwritten. A backup copy of the data in the previous version of **file 1** was saved in **file 2**.

L1300 (W) No debug information in input files
There is no debugging information in the input files. The **debug**, **sdebug**, or **compress** option has been ignored. Check whether the relevant option was specified at compilation or assembly.

L1301 (W) No inter-module optimization information in input files
No inter-module optimization information is present in the input files. The **optimize** option has been ignored. Check whether the **goptimize** option was specified at compilation or assembly.

L1302 (W) No stack information in input files
No stack information is present in the input files. The **stack** option is ignored. If all input files are assembler output files or **SYSROF**->**ELF** converted files, the **stack** option is ignored.

L1303 (W) No rts information in input files
No information in input files to generate **.rts** file. The processing will end without creating an **.rts** file.

L1304 (W) No utl information in input files
The information necessary to generate a **utl** file was not input.

L1305 (W) Entry address in "file" conflicts : "address"
Multiple files with different entry addresses are input.

L1310 (W) "section" in "file" is not supported in this tool
An unsupported section was present in **file**. **section** has been ignored.

L1311 (W) Invalid debug information format in "file"
Debugging information in **file** is not **dwarf2**. The debugging information has been deleted.

L1320 (W) Duplicate symbol "symbol" in "file"
The symbol named **symbol** is duplicated. The symbol in the first file input is given priority.

L1321 (W) Entry symbol "symbol" in "file" conflicts
Multiple object files containing more than one entry symbol definition were input. Only the entry symbol in the first file input is effective.

L1322 (W) Section alignment mismatch : "section"
Sections with the same name but different boundary alignments were input. Only the largest boundary alignment specification is effective.

L1323 (W) Section attribute mismatch : "section"
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.

L1324 (W) Symbol size mismatch : "symbol" in "file"
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.

L1325 (W) Symbol attribute mismatch : "symbol":"file"
The attribute of **symbol** in **file** does not match the attribute of the same-name symbol in other files. Check the symbol.

L1326 (W) Reserved symbol "symbol" is defined in "file"
Reserved symbol name **symbol** is defined in the **file**.

L1327 (W) Section alignment in option "aligned_section" is small : "section"
Since the boundary alignment value specified for the **aligned_section** option is 16, which is smaller than that of **section**, the option settings made for that section are ignored.

L1330 (W) Cpu type "CPU type 1" in "file" differ from "CPU type 2"
Files with different CPU types were input. Processing is continued with the CPU type assumed as H8SX.

L1400 (W) Stack size overflow in register optimization
During register optimization, the stack access code exceeded the stack size limit of the compiler. The register optimization specification has been ignored.

L1401 (W) Function call nest too deep
The number of function call nesting levels is so deep that register optimization cannot be performed.

L1402 (W) Parentheses specified in option "start" with optimization
Optimization is not available when parentheses "( )" are specified in the **start** option. Optimization has been disabled.

L1410 (W) Cannot optimize "file"-"section" due to multi label relocation operation
A section having multiple label relocation operations cannot be optimized. Section **section** in file **file** has not been optimized.

L1420 (W) "file" is newer than "profile"
file was updated after **profile**. The profile information has been ignored.

L1430 (W) Cannot generate effective bls file for compiler optimization
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.
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 program 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.

L1500 (W) Cannot check stack size
There is no stack section, and so consistency of the stack size specified by the **stack** option on compiling cannot be checked. To check the consistency of the stack size on compiling, the **goptimize** option needs to be specified on compiling and assembling.

L1501 (W) Stack size overflow : "stack size"
The stack section size exceeded the **stack size** specified by the **stack** option on

compiling. Either change the option used on compiling, or change the program so as to reduce the use of the stack.

L1502 (W) Stack size in "file" conflicts with that in another file
Different values for stack size are specified for multiple files. Check the options used on compiling.

L1510 (W) Input file was compiled with option "smap" and option "map" is specified at linkage
A file was compiled with **smap** specification. The file with **smap** specification should not be compiled with the **map** option specification in the second build processing.

P1600 (W) An error occurred during name decoding of "instance"
instance could not be decoded. The message is output using the encoding name.

L2000 (E) Invalid option : "option"
P2000 (E) Invalid option : "option"
option is not supported.

L2001 (E) Option "option" cannot be specified on command line
option cannot be specified on the command line. Specify this option in a subcommand file.

L2002 (E) Input option cannot be specified on command line
The **input** option was specified on the command line. Input file specification on the command line should be made without the **input** option.

L2003 (E) Subcommand option cannot be specified in subcommand file
The **subcommand** option was specified in a subcommand file. The **subcommand** option cannot be nested.

L2004 (E) Option "option 1" cannot be combined with option "option 2"
option 1 and **option 2** cannot be specified simultaneously.

L2005 (E) Option "option" cannot be specified while processing "process"
option cannot be specified for **process**.

L2006 (E) Option "option 1" is ineffective without option "option 2"
option 1 requires **option 2** be specified.

L2010 (E) Option "option" requires parameter
option requires a parameter to be specified.

L2011 (E) Invalid parameter specified in option "option" : "parameter"
An invalid parameter was specified for **option**.

L2012 (E) Invalid number specified in option "option" : "value"
An invalid value was specified for **option**. Check the range of valid values.

L2013 (E) Invalid address value specified in option "option" : "address"
The address **address** specified in **option** is invalid. A hexadecimal address between 0 and FFFFFFFF should be specified.

L2014 (E) Illegal symbol/section name specified in "option" : "name"
The section or symbol name specified in **option** uses an illegal character. Only alphanumerics, the underscore (_), and the dollar sign ($) may be used in section/symbol names (the leading character cannot be a number).

L2016 (E) Invalid alignment value specified in option "option" : "alignment value"
The **alignment value** specified in **option** is invalid. 1, 2, 4, 8, 16, or 32 should be specified.

L2017 (E) Cannot output "section" specified in option "option"
Part of the code in **section** specified by **option** cannot be output. Part of the instruction code in **section** has been swapped with instruction code in another section due to endian conversion. 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.
Note: The endian conversion function is available only in the RX Family CPU.

L2020 (E) Duplicate file specified in option "option" : "file"
The same file was specified twice in **option**.

L2021 (E) Duplicate symbol/section specified in option "option" : "name"
The same symbol name or section name was specified twice in **option**.

L2022 (E) Address ranges overlap in option "option" : "address range"
Address ranges **address range** specified in **option** overlap.

L2100 (E) Invalid address specified in cpu option : "address"
An invalid address was specified in the **cpu** option.

L2101 (E) Invalid address specified in option "option" : "address"
The **address** specified in **option** exceeds the address range that can be specified by the **cpu** or the range specified by the **cpu** option.

L2110 (E) Section size of second parameter in rom option is not 0 : "section"
section whose size is not zero was specified in the second parameter of the **rom** option.

L2111 (E) Absolute section cannot be specified in "option" option : "section"
An absolute address section was specified in **option**.

L2112 (E) "section 1" and "section 2" cannot mapped as ROM/RAM in "file"
section 1 and **section 2** specified in the name of **file** are not ROM/RAM-mapped.

L2113 (E) Option "rom" and internal information in the file are conflicted
Specification of the **rom** option conflicts with the internal information.

L2120 (E) Library "file" without module name specified as input file
A library file without a module name was specified as the input file.

L2121 (E) Input file is not library file : "file (module)"
The file specified by **file (module)** as the input file is not a library file.

L2130 (E) Cannot find file specified in option "option" : "file"
The file specified in **option** could not be found.

L2131 (E) Cannot find module specified in option "option" : "module"
The module specified in **option** could not be found.

L2132 (E) Cannot find "name" specified in option "option"
The symbol or section specified in **option** does not exist.

L2133 (E) Cannot find defined symbol "name" in option "option"
The externally defined symbol specified in **option** does not exist.

L2140 (E) Symbol/section "name" redefined in option "option"
The symbol or section specified in **option** has already been defined.

L2141 (E) Module "module" redefined in option "option"
The module specified in **option** has already been defined.

L2142 (E) Interrupt number "vector number" of "section" has multiple definition
Vector number definition was made multiple times in vector table **section**. Only one

address can be specified for a vector number. Check and correct the code in the source file.

L2143 (E) Invalid vector number specified: "number"
The vector number indicated by **number** is not allowed.
Check and correct the vector number specified with **#pragma special**.

L2200* (E) Illegal object file : "file"
A format other than ELF format was input.
* The error number will be shown as P2200.

L2201 (E) Illegal library file : "file"
file is not a library file.

L2202 (E) Illegal cpu information file : "file"
file is not a cpu information file.

L2203 (E) Illegal profile information file : "file"
file is not a profile information file.

L2210 (E) Invalid input file type specified for option "option" : "file (type)"
When specifying **option**, a **file (type)** that cannot be processed was input.

L2211 (E) Invalid input file type specified while processing "process" : "file (type)"
A **file (type)** that cannot be processed was input during processing **process**.

L2212 (E) "option" cannot be specified for inter-module optimization information in "file"
The option **option** cannot be used because **file** includes inter-module optimization information. Do not specify the **goptimize** option at compilation or assembly.

L2220 (E) Illegal mode type "mode type" in "file"
A file with a different **mode type** was input.

L2221 (E) Section type mismatch : "section"
Sections with the same name but different attributes (whether initial values present or not) were input.

L2223 (E) Cpu type "CPU type 1" in "file" is incompatible with "CPU type 2"
A different CPU type was input.
Since these types are incompatible in part of the specifications, even if the file is linked, correct operation cannot be guaranteed.

L2300 (E) Duplicate symbol "symbol" in "file"
There are duplicate occurrences of **symbol**.

L2301 (E) Duplicate module "module" in "file"
There are duplicate occurrences of **module**.

L2310 (E) Undefined external symbol "symbol" referenced in "file"
An undefined symbol **symbol** was referenced in **file**.

L2311 (E) Section "section 1" cannot refer to overlaid section : "section 2"-"symbol"
A symbol defined in **section 1** was referenced in **section 2** that is allocated to the same address as **section 1** overlaid. **section 1** and **section 2** must not be allocated to the same address.

L2320 (E) Section address overflowed out of range : "section"
The address of **section** exceeds the usable address range.

L2321 (E) Section "section 1" overlaps section "section 2"
The addresses of **section 1** and **section 2** overlap. Change the address specified by the **start** option.

L2322 (E) Section size too large: "section"
The size of **section** is too large. The size of a **$TBR** section must be 1024 bytes or less.

L2323 (E) Section "section 1 (address range)" overlaps with section "section 2 (address range)" in physical space
section 1 overlaps with **section 2** in the physical memory. Check the addresses of the sections.
<address range>: <section start address> - <section end address>

L2330 (E) Relocation size overflow : "file"-"section"-"offset"
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. Ensure that the referenced symbol at the **offset** position of **section** in the source list is placed at the correct position.

L2331 (E) Division by zero in relocation value calculation : "file"-"section"-"offset"
Division by zero occurred during a relocation operation. Check for problems in calculation of the position at **offset** in **section** in the source list.

L2332 (E) Relocation value is odd number : "file"-"section"-"offset"
The result of the relocation operation is an odd number. Check for problems in calculation of the position at **offset** in **section** in the source list.

L2340 (E) Symbol name "file"-"section"-"symbol..." is too long
The number of characters comprising **symbol** in **section** exceeds the translation limit of the assembler.
When outputting a symbol address file, make sure that the number of characters comprising the symbol name does not exceed the translation limit of the assembler.

L2400 (E) Global register in "file" conflicts : "symbol", "register"
Another symbol has already been allocated to a global register specified in **file**.

L2401 (E) near8, near16 symbol "symbol" is outside near memory area
symbol is not allocated in the **near8** or **near16** range. Either change the **start** specification, or remove the **near** specifier at compilation, so that correct address calculations can be made.

L2402 (E) Number of register parameter conflicts with that in another file : "function"
Different numbers of register parameters are specified for **function** in multiple files.

L2403 (E) Fast interrupt register in "file" conflicts with that in another file
The register number specified for the fast interrupt general register in **file** does not match the settings in other files. Correct the register number to match the other settings and recompile the code.

L2404 (E) Base register "base register type" in "file" conflicts with that in another file
The register number specified for **base register type** in **file** does not match the settings in other files. Correct the register number to match the other settings and recompile the code.

L2405 (E) Option "compile option" conflicts with that in other files
Specification of **compile option** is inconsistent between the input files.
Check and correct **compile option**.

L2410 (E) Address value specified by map file differs from one after linkage as to "symbol"
The address of **symbol** differs between the address within the external symbol allocation information file used at compilation and the address after linkage. Check (1) to (3) below.

(1) Do not change the program before or after the **map** option specification at compilation.

(2) **optlnk** 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 **optlnk** option for optimization.

(3) When the **tbr** option or **#pragma tbr** is used, optimization by the compiler may delete symbols after the **map** option specification at compilation. Disable the **map** option at compilation or disable the **tbr** option or **#pragma tbr**.

L2411 (E) Map file in "file" conflicts with that in another file
Different external symbol allocation information files were used by the input files at compilation.

L2412 (E) Cannot open file : "file"
file (external symbol allocation information file) cannot be opened. Check whether the file name and access rights are correct.

L2413 (E) Cannot close file : "file"
file (external symbol allocation information file) cannot be closed. There may be insufficient disk space.

L2414 (E) Cannot read file : "file"
file (external symbol allocation information file) cannot be read. An empty file may have been input, or there may be insufficient disk space.

L2415 (E) Illegal map file : "file"
file (external symbol allocation information file) has an illegal format. Check whether the file name is correct.

L2416 (E) Order of functions specified by map file differs from one after linkage as to "function name"
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.

L2417 (E) Map file is not the newest version: "file name"
The **.bls** file is not the latest version.

L2420 (E) "file 1" overlap address "file 2" : "address"
The address specified for **file 1** is the same as that specified for **file 2**.

P2500 (E) Cannot find library file : "file"
file specified as a library file cannot be found.

P2501 (E) "instance" has been referenced as both an explicit specialization and a generated instantiation
Instantiation has been requested of an instance already defined. For the file using **instance**, confirm that **form=relocate** has not been used to generate a relocatable object file.

P2502 (E) "instance" assigned to "file 1" and "file 2"
The definition of instance is duplicated in **file 1** and **file 2**. For the file using **instance**, confirm that **form=relocate** has not been used to generate a relocatable object file.

L3000 (F) No input file
There is no input file.

L3001 (F) No module in library
There are no modules in the library.

L3002 (F) Option "option 1" is ineffective without option "option 2"
The option **option 1** requires that the option **option 2** be specified.

L3004 (F) Unsupported inter-module optimization information type "type" in "file"
The file contains an unsupported inter-module optimization information **type**. Check if the compiler and assembler versions are correct.

P3005 (F) Instantiation loop
The instance generation process is iterating in a loop.
The input file name might match the name of another file. Change the file name so that there are no matching file names except the extension.

P3007 (F) Cannot create instantiation request file "file"
Unable to create an intermediate file for the instance generation process.
Check that the access rights of the object created folder and those beneath it are correct.

P3008 (F) Cannot change to directory "folder"
Unable to move to **folder**. Check that the folder exists.

P3009 (F) File "file" is read-only
file is read-only. Change its access right.

L3100 (F) Section address overflow out of range : "section"
The address of **section** exceeded FFFFFFFF. Change the address specified by the **start** option. For details of the address space, refer to the hardware manual of the target CPU.

L3102 (F) Section contents overlap in absolute section "section"
Data addresses overlap within an absolute address section. Modify the source program.

L3110 (F) Illegal cpu type "cpu type" in "file"
A file with a different cpu type was input.

L3111 (F) Illegal encode type "endian type" in "file"
A file with a different endian type was input.

L3112 (F) Invalid relocation type in "file"
There is an unsupported relocation type in **file**. Ensure the compiler and assembler versions are correct.

L3120 (F) Illegal size of the absolute code section : "section" in "file"
Absolute-addressing program section **section** in **file** has an illegal size. When the CPU type is RX Family in big endian, correct the size to a multiple of 4.

L3200 (F) Too many sections
The number of sections exceeded the translation limit. It may be possible to eliminate this problem by specifying multiple file output.

L3201 (F) Too many symbols
The number of symbols exceeded the translation limit. It may be possible to eliminate this problem by specifying multiple file output.

L3202 (F) Too many modules
The number of modules exceeded the translation limit. Divide the library.

L3203 (F) Reserved module name "optlnk_generates"
optlnk_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. Modify the name if it is used as a file name or a module name within a library.

L3300* (F) Cannot open file : "file"
file cannot be opened. Check whether the file name and access rights are correct.
* The error number will be shown as P3300.

L3301 (F) Cannot close file : "file"
file cannot be closed. There may be insufficient disk space.

L3302 (F) Cannot write file : "file"
Writing to **file** is not possible. There may be insufficient disk space.

L3303* (F) Cannot read file : "file"
file cannot be read. An empty file may have been input, or there may be insufficient disk space.
* The error number will be shown as P3303.

L3310* (F) Cannot open temporary file
A temporary file cannot be opened. Check to ensure the **HLNK_TMP** specification is correct, or there may be insufficient disk space.
* The error number will be shown as P3310.

L3311 (F) Cannot close temporary file
A temporary file cannot be closed. There may be insufficient disk space.

L3312 (F) Cannot write temporary file
Writing to a temporary file is not possible. There may be insufficient disk space.

L3313 (F) Cannot read temporary file
A temporary file cannot be read. There may be insufficient disk space.

L3314 (F) Cannot delete temporary file
A temporary file cannot be deleted. There may be insufficient disk space.

L3320* (F) Memory overflow
There is no more space in the usable memory within the linkage editor. Increase the amount of memory available.
* The error number will be shown as P3320.

L3400 (F) Cannot execute "load module"
load module cannot be executed. Check whether the path for **load module** is set correctly.

L3410 (F) Interrupt by user
An interrupt generated by **(Ctrl)** + **C** keys from a standard input terminal was detected.

L3420 (F) Error occurred in "load module"
An error occurred while executing the **load module**.

P3500 (F) Bad instantiation request file -- instantiation assigned to more than one file
An intermediate file for the instance generation process contains an error.
Recompile the files to be linked.

P3505 (F) Corrupted template information file or instantiation request file
An intermediate file for the template process or that for the instance generation process contains an error.
Do not edit these files.

L4000* (–) Internal error : ("internal error code") "file line number" / "comment"
An internal error occurred during processing by the optimizing linkage editor. 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.
* The error number will be shown as P4000.

Section 14 Translation Limits

14.1 Translation Limits of Compiler

Table 14.1 shows the translation limits of the compiler.

Source programs must be created to fall within these translation limits.

Table 14.1 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
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

No.	Classification	Item	Translation Limit
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

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

14.2 Translation Limits of Assembler

Table 14.2 shows the translation limits of the assembler.

Table 14.2 Translation Limits of Assembler

No.	Item	Translation Limit
1	Number of characters in one line	8190
2	Symbol length	Number of characters in one line*
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*
10	Number of characters in a file name	Number of characters in one line*
11	Number of characters in an environment variable setting	2048 bytes
12	Number of macro definitions	65535

Note: * The limit may become a smaller value depending on the string length specified in the same line.

Section 15 Usage Notes

This section provides notes for using this compiler package.

15.1 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.

<Example 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)
{...}
```

<Example 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 together */
 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>

```
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} ($\varepsilon = 2^{-33}$) single precision 2^{-21} ($\varepsilon = 2^{-19}$)
Absolute error for asin(1.0 – ε)	double precision 2^{-39} ($\varepsilon = 2^{-28}$) single precision 2^{-21} ($\varepsilon = 2^{-16}$)

(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>

```
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.

15.2 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
}
```

15.3 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 four options **cpu**, **endian**, **base**, and **fint_register** should be specified in the same way in the compiler, assembler, and library generator.

(b) The options in section 2.5, Microcontroller Options, except for the options in (a), must be specified in the same way in the compiler and library generator.

15.4 Compatibility with an Older Version or Older Revision

The effect of the compatibility regarding a version change or revision change is described here.

15.4.1 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 15.1.

Table 15.1 List of Intrinsic Functions Whose Type is Changed

				Changed Contents	
No.	**Item**	**Specification**	**Function**	**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 C5167(W) in V.1.01, but this warning can be avoided by deleting the cast to correct the type.

<Examples>

[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 C5167(W) 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 L3100(F) 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.

<Examples>

[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.

Section 16 Appendix

16.1 S-Type and HEX File Formats

This section describes the S-type files and HEX files that are output by the optimizing linkage editor.

16.1.1 S-Type File Format

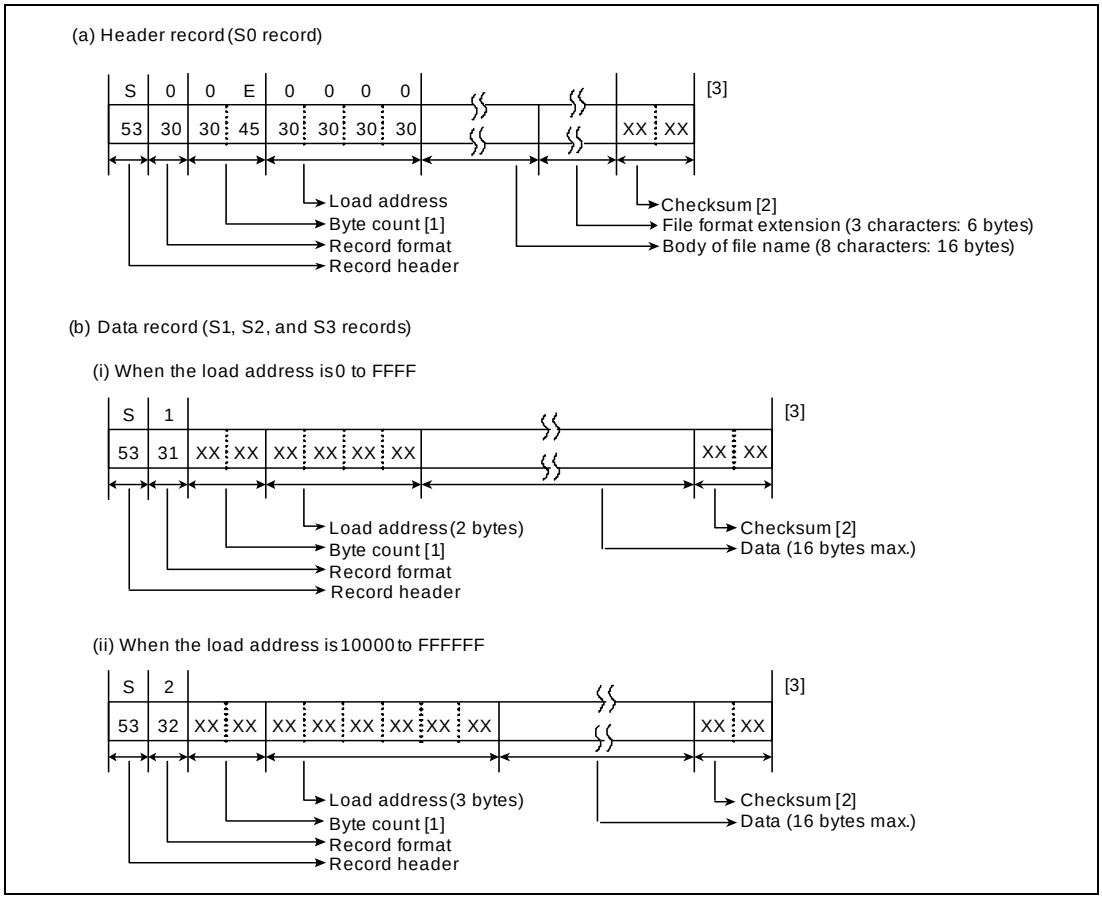
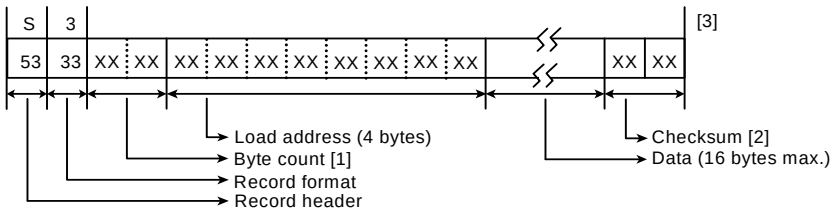


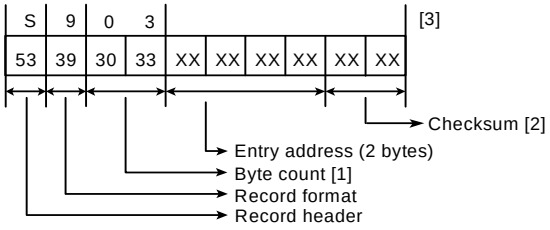
Figure 16.1 S-Type File Format

(iii) When the load address is 1000000 to FFFFFFFF

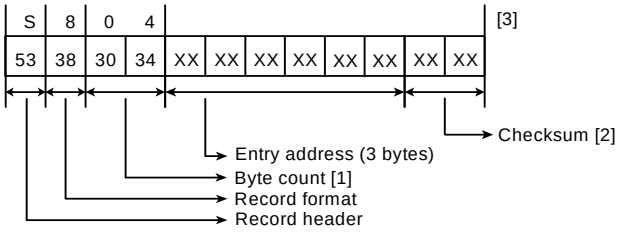


(c) End record (S9, S8, and S7 records)

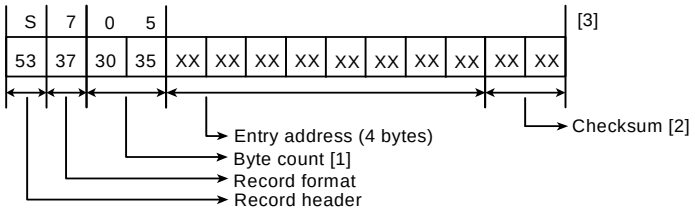
(i) When the entry address is 0 to FFFF



(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.

Figure 16.1 S-Type File Format (cont)

16.1.2 HEX File Format

The execution address of each data record is obtained as described below.

- Segment address
  (Segment base address << 4) + (Address offset of the data record)
- Linear address
  (Linear base address << 16) + (Address offset of the data record)

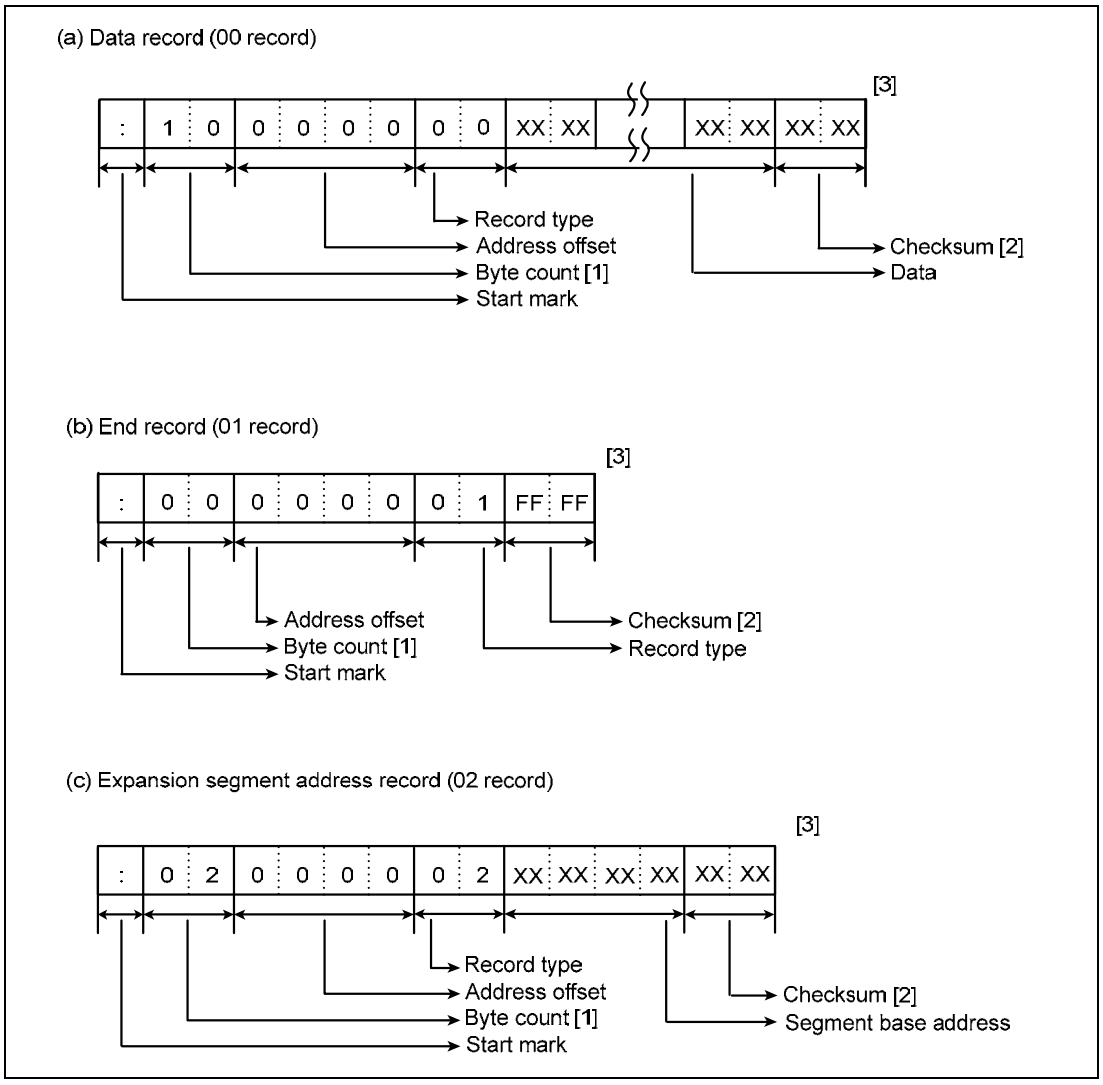
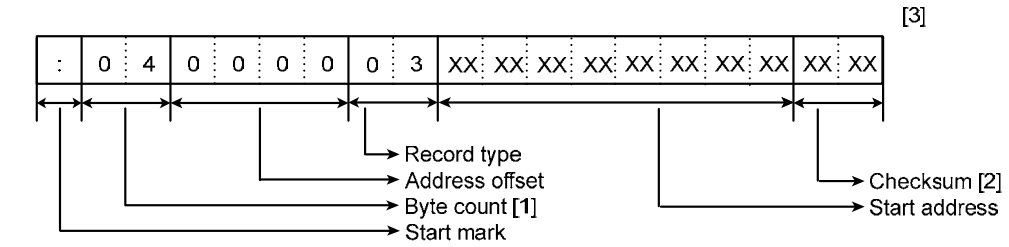
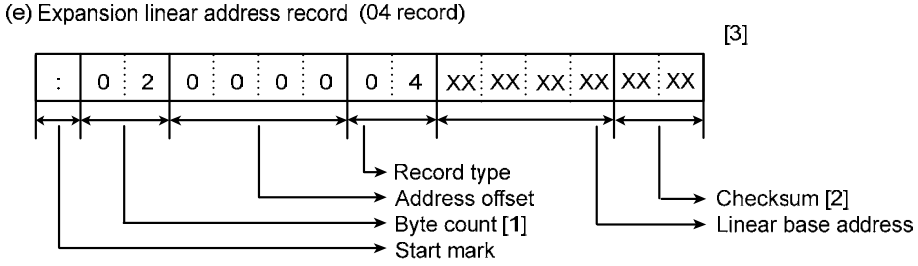


Figure 16.2 HEX File Format

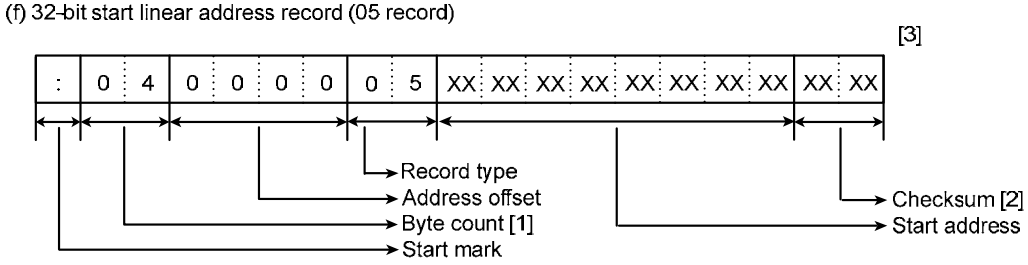
(d) Start address record (03 record)



(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.

Figure 16.2 HEX File Format (cont)

16.2 ASCII Code List

Table 16.1 ASCII Code List

Lower 4 bits	Upper 4 bits							
	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	–	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

RX Family C/C++ Compiler, Assembler, Optimizing Linkage Editor
Compiler Package (V.1.01)
User's Manual

Publication Date: Feb 23, 2011 Rev.1.00

Published by: Renesas Electronics Corporation

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
7F, No. 363 Fu Shing North Road Taipei, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

© 2010 Renesas Electronics Corporation and Renesas Solutions Corp. All rights reserved.

Colophon 1.0

RX Family C/C++ Compiler, Assembler, Optimizing Linkage Editor Compiler Package V.1.01 User's Manual

Renesas Electronics Corporation

R20UT0570EJ0100

The RX Family C/C++ Compiler V.1.02 Release 01 Notes on Usage Plus Corrections and Additions to Features Covered in the User's Manual

This document contains notes on usage of the RX Family C/C++ compiler V.1.02 Release 01 plus corrections and new features to be added to the bundled user's manual (R20UT0570EJ0100). Please read the document carefully while consulting the corresponding parts of the user's manual.

1. Notes on Usage

1.1 Note on a Case of the C1804 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 C1804(W) message. In compilation of C++ or EC++, the int_to_short option will be invalid.

[NOTE]

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.

1.2 Note on using MVTC or POPC instructions

[Assembler]

In the assembly language, the program counter (PC) cannot be specified for MVTC or POPC instructions.

1.3 Note on 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 in your debugging. If you would like to set a breakpoint at the function entrance, set the breakpoint via the Label window or at the prologue code of the function.

1.4 Note on File Names

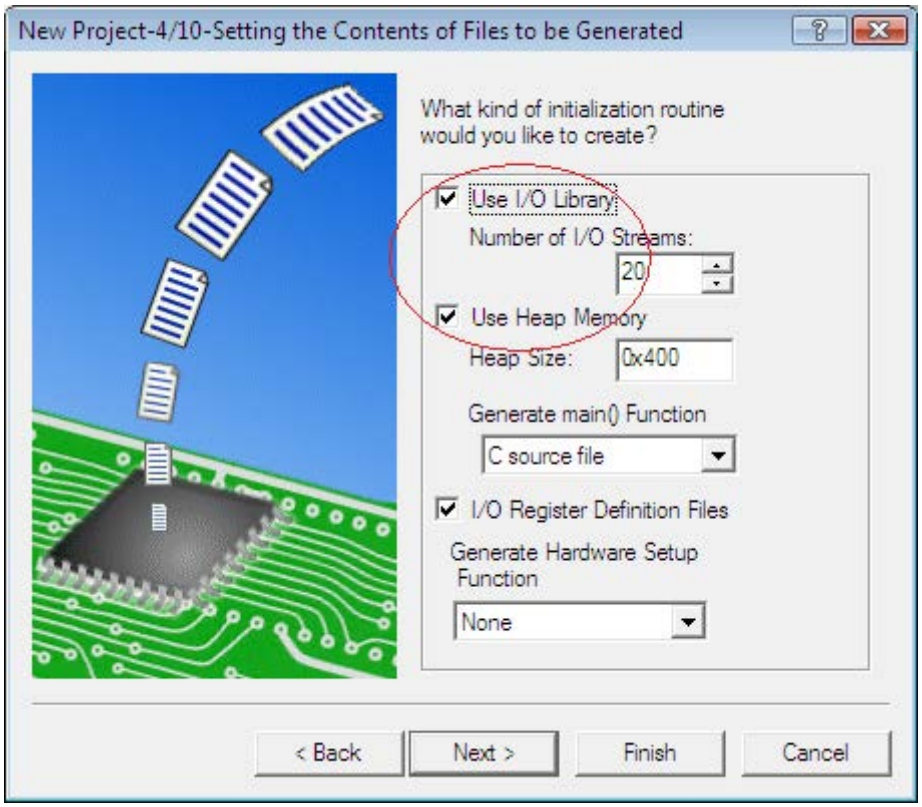
[Optimizing linkage editor]

File names must not include parentheses "(" and ")" because these characters are used for specification of options for the optimizing linkage editor.

1.5 Note on Using the I/O Library

[High-Performance Embedded Workshop - Generating Projects]

Tick [Use Heap Memory] if [Use IO Library] has been ticked for generation of a project.



If [Use I/O Library] is ticked and [Use Heap Memory] is not, the following message will be output.

```
L2310 (E) Undefined external symbol "_sbrk" referenced in "xgetmem"
```

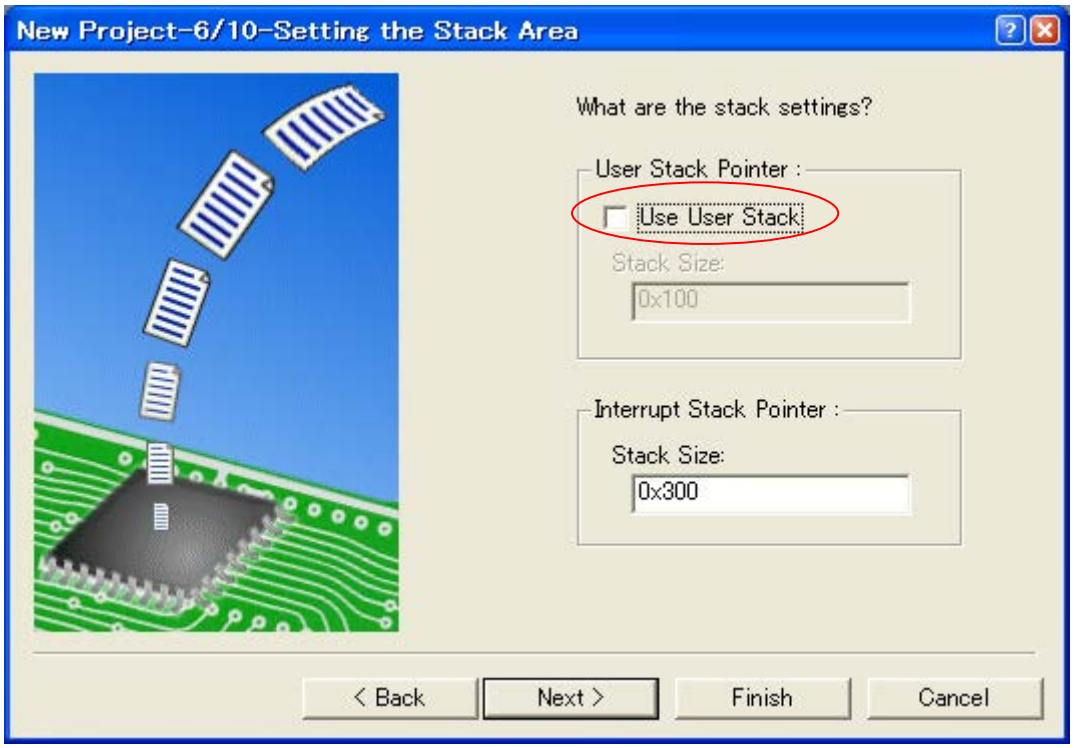
When you have encountered this problem, you should add the following C program to your project.

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

#define HEAPSIZE 0x400

signed char  *sbrk(size_t size);

union HEAP_TYPE {
    signed long  dummy ;
    signed char  heap[HEAPSIZE];
};

static union HEAP_TYPE heap_area ;

/* End address allocated by sbrk     */
static signed char *brk=(signed char *)&heap_area;

signed char  *sbrk(size_t size)
{
    signed char  *p;

    if(brk+size > heap_area.heap+HEAPSIZE){
        p = (signed char *)-1;
    }
    else {
        p = brk;
        brk += size;
    }
    return p;
}
```

1.6 How to Prevent the Creation of User Stack

[High-Performance Embedded Workshop - Generating Projects]

The [Use User Stack] checkbox is selected by default (i.e., user stack will be created) in the process of creating a new project in the High-Performance Embedded Workshop. If you do not wish to use the user stack, deselect the [Use User Stack] checkbox.



1.7 Changes in the Default Stack Sizes and Processor Mode

The default settings for the two items given below in cases where a new project is created in the IDE have been changed for V.1.01. Note, however, that the existing settings in each project that has been created with V.1.00 remain the same even if the project is used with V.1.01.

(1) Stack sizes

The default sizes of the user stack and interrupt stack are now 256 (0x100) and 768 (0x300) bytes, respectively.

(2) Processor mode

The startup program has been modified to prevent the CPU from entering the user mode by default. Thus function main is executed in the supervisor mode. Since this type of execution does not use user stack, you can prevent the creation of the user stack as described in section 1.6.

1.8 Using the Standard Library
(for Customers Who Have Upgraded the Compiler from V.1.01)

If you were using the standard library with V.1.01 Release 00 and then upgraded the compiler to V.1.02 Release 00 or newer, change environment variable TMP_RX to a different directory from that used with V.1.01 Release 00. This is necessary because the library generator, lbgrx, for V.1.01 Release 00 and later versions stores the intermediate results of creating a library in the directory indicated by TMP_RX and reuses these on the next occasion a library is created.

If you do not change the directory, the codes created by the older compiler remain in the standard libraries generated by lbgrx.

Follow the procedure below to use the standard library in the High-performance Embedded Workshop.

[Procedure to be Taken to Use the Standard Library in the High-performance Embedded Workshop]

Go through steps (1) to (3) once after upgrading the compiler to V.1.02 Release 00.

(1) Open the command prompt (the following steps require operations at the command prompt).

(2) Execute `dir %TEMP%¥*.pgl` at the command prompt and check that one or more files with a name that is a combination of numbers and letters and with the extension .pgl appears.

Result of executing `dir %TEMP%¥*.pgl` (example):

```
2011/08/09 15:47         825,346 80000400801000002 25a40409694ab0200000000.pgl
```

(3) Use the del command to delete each of the .pgl files that appeared in step (2).

Using the del command to delete one file (example):

```
del %TEMP%¥80000400801000002 25a40409694ab0200000000.pgl
```

[Remark]

If there are no .pgl files other than those that appeared in step (2), you can delete all of the .pgl files at once.

Deleting all .pgl files by using a command (example):

```
del %TEMP%¥*.pgl
```

1.9 Restrictions on the PIC/PID Function (V.1.01 or Later)

1.9.1 pic and pid Options

When a standard library is created by the library generator (lbgrx) with the pic or pid option specified, the following warning may appear once or more.

```
C1301 (W) "-pic" option ignored   (When the pic option has been specified)
C1301 (W) "-pid" option ignored   (When the pid option has been specified)
```

Despite the warning, the created standard library has no problems.

1.9.2 nouse_pid_register Option

If you are using the PID function and the nouse_pid_register option is effective for all program files of the master program, an error will occur when either of the following items is assembled.

(1) A program in which an address is assigned to a PID register

(2) Standard library (set for the library generator 'lbgrx')

[For restriction of (1):]

First, take out a function of setting PID register and put it into an independent C or assembly file.

Next, compile or assemble this file without the nouse_pid_register option.

Finally, link it and other master files together.

[For restriction of (2):]

Select either (a) or (b).

(a) Include a standard library in application not a master

* Using the jump table is not necessary. (Also see chapter 8.4.2(2)).
* Generate a standard library with setting the pid option for the library generator 'lbgrx'.

(b) Include a standard library in a master

(i) Generate a standard library without setting the pid option for the library generator 'lbgrx'.

(ii) As shown in the example below, change each JMP R14 statement of all entries in the jump table before use.

Example: Changing _printf entry

```
Before:
   _printf:
      MOV.L     #0ffff90cfH,R14  ; Address 0ffff90cfH is an example
      JMP       R14
After:
   _printf:
      MOV.L     #0ffff90cfH,R14
      PUSH.L    R13       ; PID register is R13 in this case
      JSR       R14
      POP       R13       ; PID register is R13 in this case
      RTS
```

1.10 Note on Using a Certain math.h Function (frexp, ldex, scalbn or remquo) in C++ Language (including EC++).

Compiling a C++/EC++ program that uses a certain math.h function (frexp, ldexp, scalbn or remquo) with an int-type argument generates an infinite-loop object.

Conditions:

This problem occurs when both (1) and (2) are satisfied.

(1) The program is in C++ or the lang=cpp option is effective.

(2) math.h is included and any of the following functions is called.

(a) frexp(double, long*) with the 'int *' type second argument (except when the first argument is float-type and the dbl_size=8 option is effective)

(b) ldexp(double, long) with the 'int *' type second argument (except when the first argument is float-type and the dbl_size=8 option is effective)

(c) scalbn(double, long) with the 'int *' type second argument (except when the first argument is float-type and the dbl_size=8 option is effective)

(d) remquo(double, double, long*) with the 'int *' type third argument (except when both the first and second arguments are float-type and the dbl_size=8 option is effective)

Examples:

```
file.cpp:
// Example of compiling C++ source that generates an infinite loop
#include <math.h>
double d1,d2;
int  i;
void func(void)
{
   d2 = frexp(d1, &i);
}

Command Line:
ccrx -cpu=rx600 -output=src file.cpp


file.src: Example of the generated assembly program
_func:
   ; ...(Omitted)
   BSR      __$frexp__tm__2_f__FZ1ZPi_Q2_21_Real_type__tm__4_Z1Z5_Type   ;
Calling substitute function of frexp
   ; ...(Omitted)

__$frexp__tm__2_f__FZ1ZPi_Q2_21_Real_type__tm__4_Z1Z5_Type:
L11:
   BRA   L11  ; Calls itself ==> infinite loop
```

Countermeasures:

Select one of the following ways to avoid the problem.

(1) Compile the program with the lang=c or lang=c99 option.

(2) Change int or int * into long or long *.

(3) Append the following declarations to each function that is being used.

```
/* For the frexp function */
static double frexp(double x, int *y)
{ long v = *y; double d = frexp(x,&v); *y = v; return (d); }
/* For the ldexp function */
static double ldexp(double x, int y)
{ long v = y; double d = ldexp(x,v); return (d); }
/* For the scalbn function */
static double scalbn(double x, int y)
{ long v = y; double d = scalbn(x,v); return (d); }
/* For the remquo function */
static double remquo(double x, double y, int *z)
{ long v = *z; double d = remquo(x,y,&v); *z = v; return (d); }
```

Example of (2):

```
Changing file.cpp:
#include <math.h>
double d1,d2;
int  i;
void func(void)
{
   long x = i;       /* Accept as long type temporarily */
   d2 = frexp(d1, &x);  /* Call with long type argument */
   i = x;           /* Set the result for variable 'i' */
}
```

Example of (3):

```
Changing file.cpp:
#include <math.h>
/* Append declaration */
static inline double frexp(double x, int *y)
{ long v = *y; double d = frexp(x,&v); *y = v; return (d); }
double d1,d2;
int  i;
void func(void)
{
   d2 = frexp(d1, &i);
}
```

1.11 Building Projects Converted from V.1.00 to V1.01 -- (Warning L1120 and Error L3100 on Section L)

If you convert a High-performance Embedded Workshop project created with compiler package V.1.00 Release 02 or an early version to make it compatible with V.1.01 Release 00 or later, the following warning and error message may appear.

```
L1120 (W) Section address is not assigned to "L"
L3100 (F) Section address overflow out of range : "L"
```

In such a case, select (a) or (b) given in section 2, “■ (Page 977) 15.4.1 Compatibility with V.1.00 / (2) Adding Section L (section Option and Start Option)” in this document.

Remark:

If you have used the High-performance Embedded Workshop included in this compiler package to convert a project created with compiler package V.1.00 Release 02 or an early version (RX Toolchain 1.0.0.0 to 1.0.0.2) to make it compatible with V.1.01 Release 00 or later (RX Toolchain 1.1.0.0), this problem does not occur because (b) is automatically done by the IDE.

2. Corrections in the User's Manual

■(Page 12) 2.1 Source Options / preinclude option / Description

[Correction]

Before:

If there is more than one folder specified by the **include** option, search is performed in turn starting from the leftmost folder.

Now:

If there is more than one folder specified by the **preinclude** option, search is performed in turn starting from the leftmost folder.

■(Page 15) 2.1 Source Options / change_message option / Remarks

[Addition]

This option is not usable to control the level of MISRA2004 detection messages (labeled M) that appear when the **misra2004** option has been specified.

■(Page 26) 2.2 Object Options / stuff,nostuff option / Description

[Correction]

Before:

The data contents allocated to each section are output in the order they were defined.

Now:

The data contents allocated to each section are output in the order they were defined, except that variables that do not have the initial value are output after those that have the initial value in section C.

■(Page 69) 2.5 Microcontroller Options / base option / Description

[Correction]

Before:

When <address value>=<register C> is specified, accesses to an area within 64Kbytes to 256Kbytes from the address value are performed relative to the specified register C.

Now:

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.

■(Page 283) Table 8.8 Rules for Specifying PIC/PID Function Options in Master / nouse_pid_register option / For Compilation

[Correction]

Before:

Can be specified

Now:

Can be specified except the standard library and setting PID register of the startup program.

■(Page 283) Table 8.8 Rules for Specifying PIC/PID Function Options in Master / nouse_pid_register option / Conditions on Setting the Option for Linkable Objects

[Correction]

Before:

nouse_pid_register must be specified

Now:

No conditions

■(Page 284) Table 8.10 Rules for Combinations of PIC/PID Function Options between Master and Application / Options in Master

[Correction]

Before:

nouse_pid_register is necessary

Now:

nouse_pid_register is necessary if application calls functions of master

■(Page 330) Table 9.20 #pragma Extension Specifiers and Keywords

[Addition]

No.	**Target**	**#pragma Extension Specifier*[1]**	**Function**
15	C99 standard	#pragma STDC CX_LIMITED_RANGE flag #pragma STDC FENV_ACCESS flag #pragma STDC FP_CONTRACT flag	Changes the state of the system*[3]

Notes: 3. When compilation proceeds with lang=c99 specified, the compiler only checks C99 grammars and ignores the code itself.

■(Page 335) 9.2.1 #pragma Extension Specifiers and Keywords / #pragma stacksize / Remarks

[Addition]

For <constant>, specify a value from 4 to 2147483644 (0x7ffffffc).

■(Page 351) 9.2.1 #pragma Extension Specifiers and Keywords / #pragma pack, unpack, packoption / Remarks

[Deletion]

The structure or class member for #pragma pack is specified cannot be accessed using a pointer (including an access within a member function using a pointer).

Example:

```
#pragma pack
struct st {
    char x;
    int y;
} ST;
int *p=&ST.y;    /* The ST.y address may be an odd value. */
void func(void) {
    ST.y=1;       /* Can be accessed correctly. */
    *p=1;         /* Cannot be accessed correctly in some cases. */
}
```

■(Page 448) 9.3 C/C++ Libraries / (8) <math.h> / ldexp functions / Example

[Correction]

Before:

int f;

Now:

long f;

■(Page 463) 9.3 C/C++ Libraries / (8) <math.h> / scalbn functions / Example

[Correction]

Before:

int e;

Now:

long e;

■(Page 555) 9.3.1 Standard C Libraries / (13) <stdlib.h> / mbstowcs

[Addition]

Type	Definition Name	Description
Function	mbstowcs	Converts a multibyte string to a wide string. For details, see page 673.

■(Page 555) 9.3.1 Standard C Libraries / (13) <stdlib.h> / wcstombs

[Addition]

Type	Definition Name	Description
Function	wcstombs	Converts a wide string to a multibyte string. For details, see page 674.

■(Page 803) 10.3.1 Address Directives / .BLKB / Remarks

[Addition]

The maximum value specifiable for an operand is 7FFFFFFFH.

■(Page 804) 10.3.1 Address Directives / .BLKW / Remarks

[Addition]

The maximum value specifiable for an operand is 3FFFFFFFH.

■(Page 805) 10.3.1 Address Directives / .BLKL / Remarks

[Addition]

The maximum value specifiable for an operand is 1FFFFFFFH.

■(Page 806) 10.3.1 Address Directives / .BLKD / Remarks

[Addition]

The maximum value specifiable for an operand is 0FFFFFFFH.

■(Page 921) 11.2 List of Messages / C6373, C6374, C6375

[Deletion]

```
C6373 (W) Implicit conversion of a 64-bit integral type to a smaller
integral type (potential portability problem)
C6374 (W) Explicit conversion of a 64-bit integral type to a smaller
integral type (potential portability problem)
C6375 (W) Conversion from pointer to same-sized integral type (potential
portability problem)
```

■(Page 965) 14.2 Translation Limits of Assembler / Table 14.2 /
No.1 / Number of characters in one line

[Correction]

Before:

8190

Now:

32760

■(Page 977) 15.4.1 Compatibility with V.1.00 / (2) Adding Section L (section Option and Start Option)

[Correction] Corrected descriptions are as follows.

(2) Adding Section L (section Option and Start Option)

V.1.01 newly provides section L, in which literal areas such as string literals are to be output. Using section L improves the code efficiency in cases where the **map** or **base** option is specified for section C. Since section L is enabled by default, however, programs that use literal areas contain more sections compared to V.1.00.

The optimizing linkage editor may output address error L3100(F) in some cases because section L is added to the end of other sections at linkage unless otherwise specified.

```
L3100 (F) Section address overflow out of range : "L"
```

To avoid this error, adopt either method (a) or (b) given below.

We recommend (a) in terms of code efficiency. Only choose (b) if you do not want to change the configuration of sections.

(a) Add L to the section sequence specified with the Start option of the optimizing linkage editor at linkage.

• From the command line

<Examples>

[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
```

• From the High-performance Embedded Workshop

1) Select [Build -> RX Standard Toolchain].

2) Click on the [Link/Library] tab and select [Section] from the [Category] menu. Then select

[Section] from the [Show entries for:] menu.

3) Select an address from the list where you wish to allocate section L and click on the [Modify] or [Add] button.

(b) Select -section=L=C at the time of compilation or building a library.

• From the command line

Specifying -section=L=C for the **ccrx** or **lbgrx** command, respectively, at the time of compilation or building a library changes the output destination of the literal area to section C and thus makes a configuration of sections compatible with V.1.00.

Note that this method may affect code efficiency compared to method (a) of changing the Start option at linkage.

• From the High-performance Embedded Workshop

1) Select [Build -> RX Standard Toolchain].

2) Click on the [C/C++] tab.

3) Select [Object] from the [Category] menu.

4) Click on the [Details…] button to open the [Object details] dialog box.

5) Select [Literal section (L)] from the pull-down menu and enter “C” in the text box below. Then click on the [OK] button to close the dialog box.

6) Click on the [Standard Library] tab and repeat steps 3 to 5.

3. Additional Features

V.1.02 Release 00 provides the additional features listed below.

3.1 Source Options [Section 2.1 of the user's manual]

The following source options have been newly added.

No.	Option	Dialog Menu	Description	
11	-misra2004={ all | apply=<list of rule numbers> | ignore=<list of rule numbers> | required | required_add=<list of rule numbers> | required_remove=<list of rule numbers> | <filename> } <list of rule numbers>: <rule number>[,<rule number>,...]		Checks the source code against the MISRA-C: 2004 rules.	New
12	-ignore_files_misra= <filename>[,<filename>,...]		Selects files that will not be checked against the MISRA-C: 2004 rules.	New
13	-check_language_extension		Enables complete checking against the MISRA-C: 2004 rules for parts of the code where this would otherwise be suppressed due to use of an extended specification.	New

misra2004

Format: -misra2004 = {
all
| apply=<rule number>[,<rule number>,...]
| ignore=<rule number>[,<rule number>,...]
| required
| required_add=<rule number>[,<rule number>,...]
| required_remove=<rule number>[,<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 a decimal value 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): `C6700 (M) Rule number: Message`

Remarks: If a single option is specified more than once in the command line, only the last specification is valid.

When the number of an unsupported rule is specified for <rule number>, the compiler detects error C6703(F) and stops the processing.

When the file specified in **misra2004=<filename>** cannot be opened, the compiler detects error C6701(F). When rule numbers are not extractable from the specified file, the compiler detects error C6702(F). 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.

2.2 2.3
4.1 4.2
5.2 5.3 5.4 5.5 5.6
6.1 6.2 6.3 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 11.3 11.4 11.5
12.1 12.3 12.4 12.5 12.6 12.7 12.8 12.9 12.10 12.11
12.12 12.13
13.1 13.2 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
17.5
18.1 18.4
19.3 19.6 19.7 19.8 19.11 19.13 19.14 19.15
20.4 20.5 20.6 20.7 20.8 20.9 20.10 20.11 20.12

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.

ignore_files_misra

Format: -ignore_files_misra=<filename>[,<filename>,...]

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

Remarks: If a single option is specified more than once in the command line, all specifications are valid.

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

<filename> is ignored when the specified file is not to be compiled.

check_language_extension

Format: -check_language_extension

Description: This option enables complete checking against the MISRA-C: 2004 rules for parts of the code where it would otherwise be suppressed due to individual extensions from the C/C++ language specification.

With the default **misra2004** option, the compiler does not proceed with checking against the MISRA-C: 2004 rules under the condition given below. To enable complete checking, specify the **check_language_extension** option.

Condition:

The function has no prototype declaration (rule 8.1) and **#pragma entry** or **#pragma interrupt** is specified for it.

Example:

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

Function **vfunc**, for which **#pragma interrupt** is specified, has no prototype declaration. Even when this function is compiled with **-misra2004=all** specified, 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** option has not been specified.

3.2 Object Option [Section 2.2 of the user's manual]

The following object option has been newly added.

No.	Option	Description	
7	-nouse_div_inst	Generates code in which no DIV, DIVU, or FDIV instructions are used for division and modular division.	New

nouse_div_inst

Format: -nouse_div_inst

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

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

This option is also usable for the library generator (lbgrx).

3.3 Compiler Error Level [Section 11.1 of the user's manual]

The following compiler error level has been newly added.

	Error Level	Compiler Operation	
(M)	MISRA2004 detection	Processing is continued.	New

3.4 Compiler Error Messages [Section 11.2 of the user's manual]

The following compiler error messages have been newly added.

```
C6700 (M) Rule <rule number>: <description>

C6701 (F) Cannot open rule file <filename>

C6702 (F) Incorrect description "<description>" in rule file

C6703 (F) Rule <rule number> is unsupported
```

3.5 Translation Limits of Compiler [Section 14.1 of the user's manual]

The following item has been newly added to the table of Translation Limits of Compiler.

No.	Classification	Item	Translation Limit	
32	Output file	Maximum number of characters in a line output as assembly source code	8190	New

Standard Libraries Included in RX Family C/C++ Compiler Package V.1.00 Release 01

This compiler package includes four library files (*.lib) for the RX600. You can use any of the library files if they correspond to the options that you wish to specify. Using these files shortens the time required for building.

1. Library Files

Table 1 shows the standard library files and compiler options.

Note:

The compiler options you specify should be the same as the microcontroller options defined for each of the library files listed in table 1. Otherwise these library files are not usable, so specify your compiler options in the library generator to generate your own library file.

Library File	Purposes	Optimize *2 Options	Microcontroller Options *1 *2		
			-endian	-cpu -rtti -exception -noexception	Others *3
rx600lq.lib	For the RX600 Optimization type: Speed Little endian	-speed -goptimize	-endian=little	-cpu=rx600 -rtti=on -exception	-round=nearest -denormalize=off -dbl_size=4 -unsigned_char -unsigned_bitfield -bit_order=right -unpack -fint_register=0 -branch=24
rx600ls.lib	For the RX600 Optimization type: Size Little endian	-size -goptimize			
rx600bq.lib	For the RX600 Optimization type: Speed Big endian	-speed -goptimize	-endian=big		
rx600bs.lib	For the RX600 Optimization type: Size Big endian	-size -goptimize			

Table 1 Library Files

*Notes:

*1 For details on microcontroller options, refer to section 2.5, Microcontroller Options, in the user's manual for the compiler.

*2: For confirming the option selections from the High-performance Embedded Workshop's build settings, please see the "Dialog Menu" columns of the "Table 2.7 Optimize Options" and "Table 2.9 Microcontroller Options" in the User's manual.

*3: These option selections are same from the each default of them.

2. Using the Library Files

The library files included in the compiler package must be linked in either of the ways given in sections 2.2 and 2.3.

2.1 Location of the Library Files

When the High-performance Embedded Workshop has been installed in C:\Program Files\Renesas\Hew, the library files are stored in the following location:

```
C:\Program Files\Renesas\Hew\Tools\Renesas\RX\1_0_1\lib
```

("1_0_1" indicates the version and revision number of the compiler package.)

2.2 Selecting a Library File through the High-performance Embedded Workshop

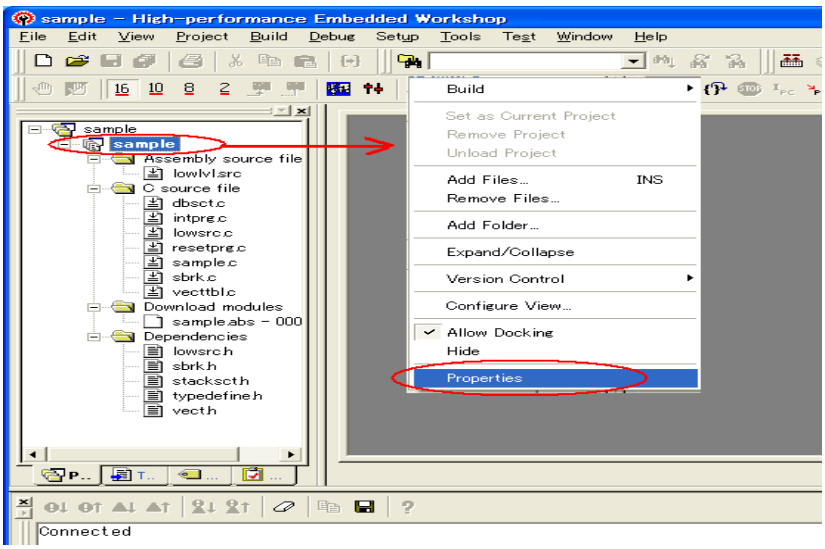
Follow the procedure below to select a library file for the project you are using.

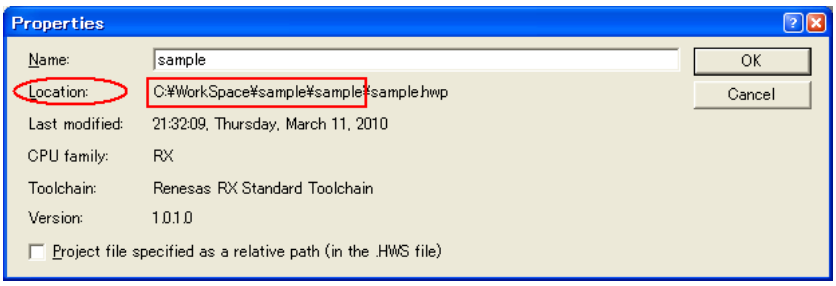
(1) Open the project.

(2) Please confirm the project setting, and select one of libraries on the Table 1 above.

(3) Check the location of the project directory.

Select a project in the [Workspace] window and right-click on it. Then select [Properties] from the popup menu and check the path displayed on the right to [Location].





The directory containing a file with extension .hwp is the project directory.

(4) Copy the library file you selected of (2), from the location given in section 2.1 to the project

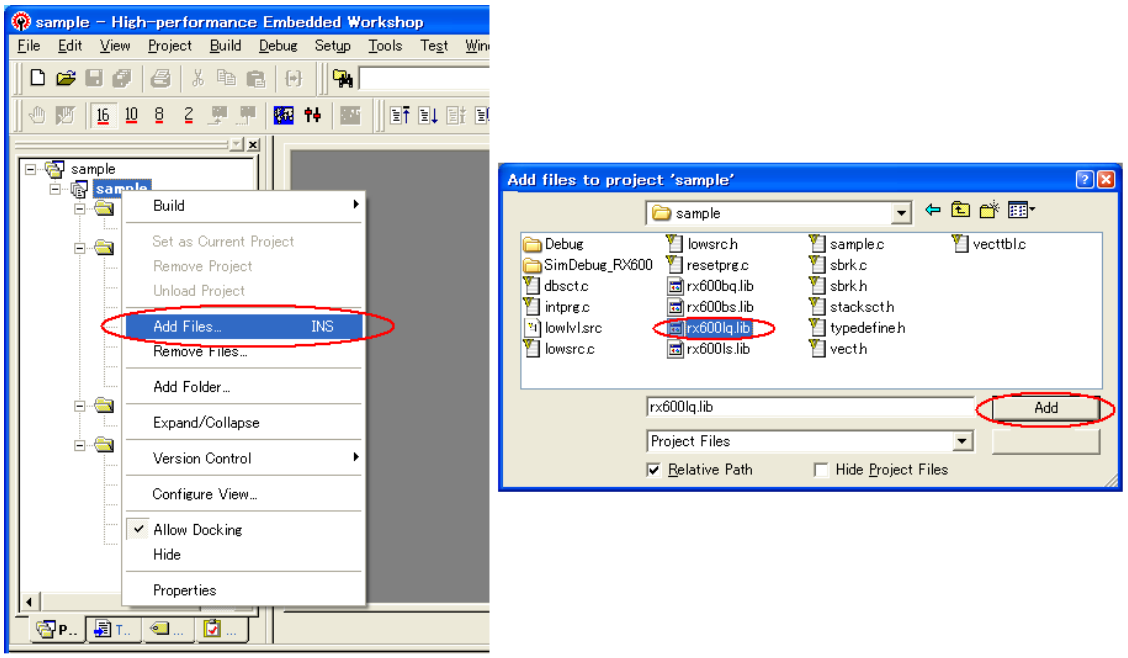
directory of (3).

[An example of the step (4) on the command prompt]

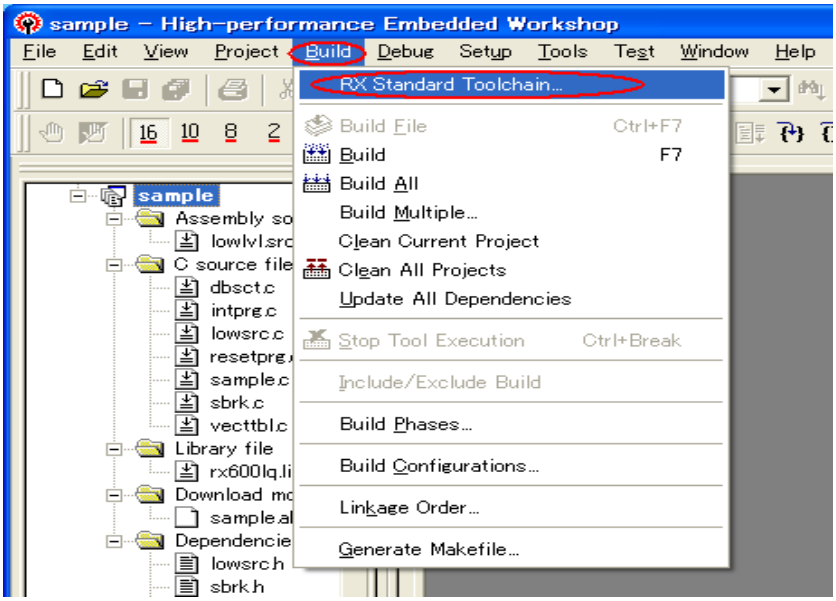
```
copy "C:\Program Files\Renesas\Hew\Tools\Renesas\RX\1_0_1\lib\rx600lq.lib" C:\WorkSpace\sample\sample
```

(5) Select a project in the [Workspace] window and right-click on it. Then select [Add files... INS] from the popup menu.

(6) Select the library file copied in step (4) and click on the [Add] button.



(7) Select [Build => RX Standard Toolchain...].

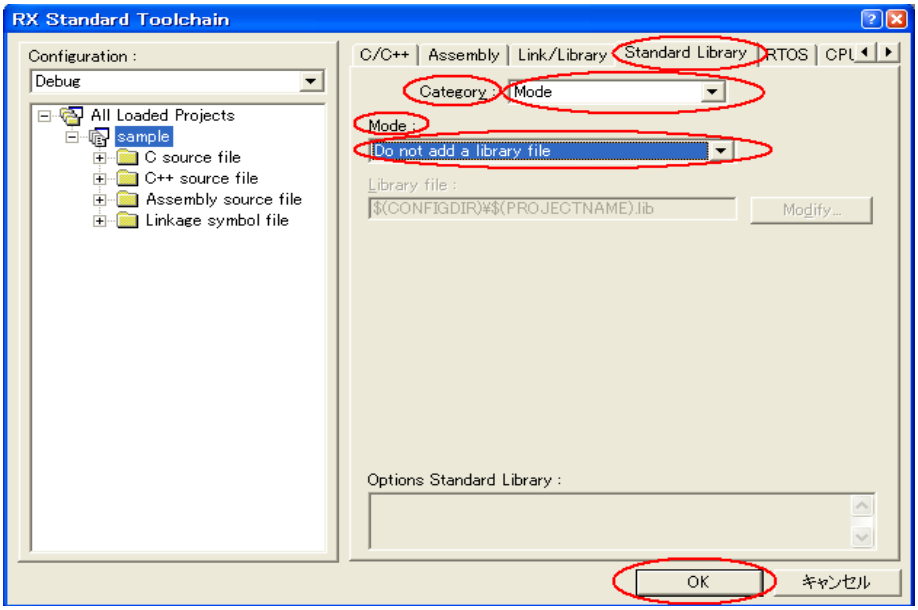


(8) Click on the [Standard Library] tab.

(9) Select [Mode] from the [Category:] pull-down menu.

(10) Select [Do not add a library file] from the [Mode:] pull-down menu.

(11) Click on the [OK] button to save the new setting.



Setting of the project is now complete.

When building of the project is executed, the library file selected in step (6) is linked.

2.3 Directly Specifying a Library File in the Optimizing Linkage Editor

Copy the library file(s) included in the package (stored in the location given in section 2.1) into a desired directory.

Then specify one of the copied library files for the Library option and start the linkage processing.