

CubeSuite+ V2.02.00

Integrated Development Environment

User's Manual: RX Coding

Target Device RX Family

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 Corp. without notice. Please review the latest information published by Renesas Electronics Corp. through various means, including the Renesas Electronics Corp. website (http://www.renesas.com).

Notice

- 1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation 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.
- 2. 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.
- 3. 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.
- 4. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics product.
- 5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

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

"High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anticrime systems; and safety equipment etc.

Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (nuclear reactor control systems, military equipment etc.). 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 for which it is not intended. 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 which the product is not intended by Renesas Electronics.

- 6. 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.
- 7. 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 systems manufactured by you.
- 8. 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.
- 9. 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. You should not use Renesas Electronics products or 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. When exporting the Renesas Electronics 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.
- 10. It is the responsibility of the buyer or distributor of Renesas Electronics products, who distributes, disposes of, or otherwise places the product with a third party, to notify such third party in advance of the contents and conditions set forth in this document, Renesas Electronics assumes no responsibility for any losses incurred by you or third parties as a result of unauthorized use of Renesas Electronics products.
- 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 majorityowned subsidiaries.
- (Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

How to Use This Manual

This manual describes the role of the CubeSuite+ integrated development environment for developing applications and systems for RX families, and provides an outline of its features.

CubeSuite+ is an integrated development environment (IDE) for RX families, integrating the necessary tools for the development phase of software (e.g. design, implementation, and debugging) into a single platform.

By providing an integrated environment, it is possible to perform all development using just this product, without the need to use many different tools separately.

Readers	This manual is intended for users who wish to understand the functions of the CubeSuite+ and design software and hardware application systems.			
Purpose	This manual is intended to give users an understanding of the functions of the CubeSuite+ to use for reference in developing the hardware or software of systems using these devices.			
Organization	This manual can be broadly divided into the following units.			
	1.GENERAL 2.FUNCTIONS 3.Compiler Language Specifications 4.ASSEMBLY LANGUAGE SPECIFICATIONS 5.LINK DIRECTIVE SPECIFICATIONS 6.Function Specifications 7.STARTUP 8.Referencing Compiler and Assembler 9.Usage Notes A.WINDOW REFERENCE			
How to Read This Manual	I It is assumed that the readers of this manual have general knowledge of electricity, logic circuits, and microcontrollers.			
Conventions	Data significance:	Higher digits on the left and lower digits on the right		
	Active low representation:	XXX (overscore over pin or signal name)		
	Note:	Footnote for item marked with Note in the text		
	Caution:	Information requiring particular attention		
	Remark: Supplementary information			
	Numeric representation:	Decimal XXXX Hexadecimal 0xXXXX		
Related Documents	The related documents indic However, preliminary version	ated in this publication may include preliminary versions. ns are not marked as such.		

Document Name		Document No.
CubeSuite+	Start	R20UT2865E
Integrated Development Environ-	V850 Design	R20UT2134E
User's Manual	RL78 Design	R20UT2684E
	78K0R Design	R20UT2137E
	78K0 Design	R20UT2138E
	RX Coding	This manual
	V850 Coding	R20UT0553E
	Coding for CX Compiler	R20UT2659E
	RL78, 78K0R Coding	R20UT2774E
	78K0 Coding	R20UT2141E
	RX Build	R20UT2998E
	V850 Build	R20UT0557E
	Build for CX Compiler	R20UT2142E
	RL78, 78K0R Build	R20UT2623E
	78K0 Build	R20UT0783E
	RX Debug	R20UT2875E
	V850 Debug	R20UT2446E
	RL78 Debug	R20UT2867E
	78K0R Debug	R20UT0732E
	78K0 Debug	R20UT0731E
	Analysis	R20UT2868E
	Message	R20UT2871E

Caution The related documents listed above are subject to change without notice. Be sure to use the latest edition of each document when designing.

All trademarks or registered trademarks in this document are the property of their respective owners.

TABLE OF CONTENTS

1.	GENERAL	. 10
1.1	Overview	10
1.2	Special Features	12
1.3	Limits	12
1.3.1	Limits of Compiler	12
1.3.2	Limits of Assembler	14
•		4 -
2.	FUNCTIONS	. 15
2.1	Variables (C Language)	15
2.1.1	Changing Mapped Areas	15
2.1.2	2 Defining Variables Used at Normal Processing and Interrupt Processing	16
2.1.3	Generating a Code that Accesses Variables in the Declared Size	16
2.1.4	Performing const Declaration for Variables with Unchangeable Initialized Data	17
2.1.5	Defining the const Constant Pointer	17
2.1.6	Referencing Addresses of a Section	18
2.2	Functions	18
2.2.1	Filling Assembler Instructions	18
2.2.2	Performing In-Line Expansion of Functions	18
2.2.3	Performing (Inter-File) In-Line Expansion of Functions	19
2.3	Using Microcomputer Functions	19
2.3.1	Processing an Interrupt in C Language	19
2.3.2	Using CPU Instructions in C Language	20
2.4	Variables (Assembly Language)	21
2.4.1	Defining Variables without Initial Values	21
2.4.2	Defining a cost Constant with an Initial Value	21
2.4.3	Referencing the Address of a Section	22
2.5	Startup Routine	22
2.5.1	Allocating Stack Areas	22
2.5.2	2 Initializing RAM	23
2.5.3	3 Transferring Variables with Initial Values from ROM to RAM	23
2.6	Reducing the Code Size	23
2.6.1	Data Structure	23
2.6.2	Local Variables and Global Variables	24
2.6.3	Offset for Structure Members	25
2.6.4	Allocating Bit Fields	27
2.6.5	Optimization of External Variable Accesses when the Base Register is Specified	28
2.6.6	Specified Order of Section Addresses by Optimizing Linkage Editor at Optimization of External Varial Accesses29	ole
2.6.7	Modularization of Functions	30

2.6.8	Interrupt	31
2.7	High-Speed Processing	32
2.7.1	Loop Control Variable	32
2.7.2	Function Interface	33
2.7.3	Reducing the Number of Loops	34
2.7.4	Usage of a Table	35
2.7.5	Branch	36
2.7.6	Inline Expansion	38
2.8	Mutual Reference between Compiler and Assembler	40
2.8.1	Referencing Assembly-Language Program External Names in C/C++ Programs	40
2.8.2	Referencing C/C++ Program External Names (Variables and C Functions) from Assembly-Languag grams40	e Pro-
2.8.3	Referencing C++ Program External Names (Functions) from Assembly-Language Programs	41
3. C	Compiler Language Specifications	. 42
3.1	Basic Language Specifications	42
3.1.1	Unspecified Behavior	42
3.1.2	Undefined Behavior	42
3.1.3	Processing System Dependent Items.	44
3.1.4	Internal Data Representation and Areas	49
3.1.5	Operator Evaluation Order	61
3.1.6	Conforming Language Specifications	62
3.2	Extended Language Specifications	62
3.2.1	Macro Names	62
3.2.2	Keywords	64
3.2.3	#pragma Directive	64
3.2.4	Using Extended Specifications	66
3.2.5	Using a Keyword.	77
3.2.6	Intrinsic Functions	78
3.2.7	Section Address Operators.	98
3.3	Modification of C Source	99
3.4	Function Calling Interface	100
3.4.1	Rules Concerning the Stack	100
3.4.2	Rules Concerning Registers	100
3.4.3	Rules Concerning Setting and Referencing Parameters	102
3.4.4	Rules Concerning Setting and Referencing Return Values	103
3.4.5	Examples of Parameter Allocation	104
3.5	List of Section Names	106
3.5.1	C/C++ Program Sections	107
3.5.2	Assembly Program Sections	109
3.5.3	Linking Sections	110
4. A	ASSEMBLY LANGUAGE SPECIFICATIONS	113

4.1	Description of Source.	113
4.1.1	Description	113
4.1.2	Names	113
4.1.3	Coding of Labels	114
4.1.4	Coding of Operation	114
4.1.5	Coding of Operands	115
4.1.6	Expression	116
4.1.7	Coding of Comments	122
4.1.8	Selection of Optimum Instruction Format	123
4.1.9	Selection of Optimum Branch Instruction	129
4.1.10	Substitute Register Names (for the PID Function)	130
4.2	Directives	131
4.2.1	Outline	131
4.2.2	Link Directives	131
4.2.3	Assembler Directives	133
4.2.4	Address Directives	135
4.2.5	Macro Directives	143
4.2.6	Specific Compiler Directives	150
4.3	Control Instructions	150
4.3.1	Outline	150
4.3.2	Assembler List Directive	150
4.3.3	Conditional Assembly Directives	151
4.3.4	Extended Function Directives	152
4.4	Macro Names.	156
4.5	Reserved Words	157
4.6	Instructions.	158
4.6.1	Address Space	158
4.6.2	Register Configuration	158
4.6.3	Processor Status Word (PSW)	160
4.6.4	Floating-Point Status Word (FPSW)	160
4.6.5	Internal State after Reset is Cleared.	160
4.6.6	Data Types	160
4.6.7	Data Arrangement	162
4.6.8	Vector Tables	163
4.6.9	Addressing Modes	165
4.6.10) Guide to This Chapter.	165
4.6.11	General Instruction Addressing	166
4.6.12	2 Instruction overview	172
4.6.13	3 Functions	172
_ .		
5. L	INK DIRECTIVE SPECIFICATIONS	/4
5.1	Section mapping	274

5.2	Section type
6. F	unction Specifications
6.1	Supplied Libraries
6.1.1	Terms Used in Library Function Descriptions
6.1.2	Notes on Use of Libraries
6.2	Header Files
6.3	Reentrant Library
6.4	Library Function
6.4.1	<stddef.h></stddef.h>
6.4.2	<assert.h></assert.h>
6.4.3	<ctype.h></ctype.h>
6.4.4	<float.h></float.h>
6.4.5	limits.h>
6.4.6	<pre><errno.h></errno.h></pre>
6.4.7	<math.h></math.h>
6.4.8	<mathf.h></mathf.h>
6.4.9	<setjmp.h></setjmp.h>
6.4.10	<stdarg.h></stdarg.h>
6.4.11	<stdio.h></stdio.h>
6.4.12	<stdlib.h></stdlib.h>
6.4.13	<string.h></string.h>
6.4.14	<complex.h></complex.h>
6.4.15	<fenv.h></fenv.h>
6.4.16	<inttypes.h></inttypes.h>
6.4.17	<iso646.h></iso646.h>
6.4.18	<stdbool.h></stdbool.h>
6.4.19	<stdint.h></stdint.h>
6.4.20	<tgmath.h></tgmath.h>
6.4.21	<wchar.h></wchar.h>
6.5	EC++ Class Libraries
6.5.1	Stream Input/Output Class Library
6.5.2	Memory Management Library
6.5.3	Complex Number Calculation Class Library
6.5.4	String Handling Class Library
6.6	Unsupported Libraries
7. S	TARTUP
7.1	Overview
7.2	File Contents
7.3	Startup Program Creation
7.3.1	Fixed Vector Table Setting
7.3.2	Initial Setting

7.3.3	Coding Example of Initial Setting Routine48	80
7.3.4	Low-Level Interface Routines	81
7.3.5	Termination Processing Routine	92
7.4	Coding Example	95
7.5	Usage of PIC/PID Function	6
7.5.1	Terms Used in this Section	6
7.5.2	Function of Each Option)6
7.5.3	Restrictions on Applications)7
7.5.4	System Dependent Processing Necessary for PIC/PID Function)7
7.5.5	Combinations of Code Generating Options)7
7.5.6	Master Startup	8
7.5.7	Application Startup	9
8.	Referencing Compiler and Assembler	3
8.1	Method for Mutual Referencing of External Names	3
8.1 9.	Method for Mutual Referencing of External Names	3 5
8.1 9. 9.1	Method for Mutual Referencing of External Names 51 Usage Notes 51 Notes on Program Coding 51	3 5
8.1 9. 9.1 9.2	Method for Mutual Referencing of External Names 51 Usage Notes 51 Notes on Program Coding 51 Notes on Compiling a C Program with the C++ Compiler 51	3 5 5 8
8.1 9. 9.1 9.2 9.3	Method for Mutual Referencing of External Names 51 Usage Notes 51 Notes on Program Coding 51 Notes on Compiling a C Program with the C++ Compiler 51 Notes on Options 51	3 5 8 8
8.1 9. 9.1 9.2 9.3 9.4	Method for Mutual Referencing of External Names 51 Usage Notes 51 Notes on Program Coding 51 Notes on Compiling a C Program with the C++ Compiler 51 Notes on Options 51 Compatibility with an Older Version or Older Revision 51	3 5 8 8 8
8.1 9. 9.1 9.2 9.3 9.4 9.4.1	Method for Mutual Referencing of External Names 51 Usage Notes 51 Notes on Program Coding 51 Notes on Compiling a C Program with the C++ Compiler 51 Notes on Options 51 Compatibility with an Older Version or Older Revision 51 Compatibility with V1.00 51	3 5 8 8 8 8
8.1 9. 9.1 9.2 9.3 9.4 9.4.1 9.4.2	Method for Mutual Referencing of External Names 51 Usage Notes 51 Notes on Program Coding 51 Notes on Compiling a C Program with the C++ Compiler 51 Notes on Options 51 Compatibility with an Older Version or Older Revision 51 Compatibility with V.1.00 51 Compatibility with V.1.01 and V.1.02 52	3 5 8 8 8 8
8.1 9. 9.1 9.2 9.3 9.4 9.4.1 9.4.2 A. V	Method for Mutual Referencing of External Names 51 Usage Notes 51 Notes on Program Coding 51 Notes on Compiling a C Program with the C++ Compiler 51 Notes on Options 51 Compatibility with an Older Version or Older Revision 51 Compatibility with V.1.00 51 Compatibility with V.1.01 and V.1.02 52 VINDOW REFERENCE 52	13 5 15 18 18 18 18 20 1
8.1 9. 9.1 9.2 9.3 9.4 9.4.1 9.4.2 A. V A.1	Method for Mutual Referencing of External Names 51 Usage Notes 51 Notes on Program Coding 51 Notes on Compiling a C Program with the C++ Compiler 51 Notes on Options 51 Compatibility with an Older Version or Older Revision 51 Compatibility with V.1.00 51 Compatibility with V.1.01 and V.1.02 52 VINDOW REFERENCE 52 Description 52	13 5 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8

1. GENERAL

This chapter introduces the processing of compiling performed by the RX family C/C++ compiler, and provides an example of program development.

1.1 Overview

The build tool is comprised of components provided by CC-RX. It enables various types of information to be configured via a GUI tool, enabling you to generate load module file or library file from your source files, according to your objectives. CC-RX is comprised of the four executable files listed below.

- (1) ccrs: Compile driver
- (2) asrx: Assembler Optimizer
- (3) rlink: Optimizing linkage editor
- (4) Ibgrx: Library generator

Figure 1.1 illustrates the CC-RX processing flow.







RENESAS

1.2 Special Features

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

- Language specifications in accordance with ANSI standard The C, C99, and C++ language specifications conform to the ANSI standard. Coexistence with prior C language specifications (K&R specifications) is also provided.
- (2) Advanced optimization Code size and speed priority optimization for the C compiler are offered.
- (3) Improvement to description ability C language programming description ability has been improved due to enhanced language specifications.
- (4) High portability
 The single CC-RX supports all microcontrollers. This makes it possible to use a uniform language specification, and facilitates porting between microcontrollers.
 In addition, the industry-standard DWARF2/3 format is used for debugging information.
- Multifunctional Static analysis and other functionality is provided via linking between CubeSuite+.

1.3 Limits

1.3.1 Limits of Compiler

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

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

Table 1.1Translation Limits of Compiler



No.	Classification	Item	Translation Limit
12	Declaration	Number of function definitions	Unlimited
13		Number of external identifiers used for external linkage	Unlimited
14		Number of valid internal identifiers used in one function	Unlimited
15		Number of pointers, arrays, and function declarators that qualify the basic type	16
16		Number of array dimensions	6
17		Size of arrays and structures	2147483647 bytes

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
32		Maximum size of each section	4294967295 bytes
33	Output files	Maximum number of characters per line of assembly source code that can be output	8190

Notes 1. For details, refer to section 7.3.2 Initial Setting.

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

1.3.2 Limits of Assembler

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

Table 1.2 Translation Limits of Assembler

No.	Item	Translation Limit
1	Number of characters in one line	32760
2	Symbol length	Number of characters in one line*
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	0FFFFFFFH 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.



2. FUNCTIONS

This chapter describes programming methods and the usage of extended functions for effective use of the RX family.

2.1 Variables (C Language)

This section describes variables (C language).

2.1.1 Changing Mapped Areas

The defaults for the mapped sections of variables are as follows:

- Variables without initial values: Sections B, B_2, and B_1

- Variables with initial values: Sections D, D_2, and D_1 (ROM) and sections R, R_2, and R_1 (RAM)

- const variables: Sections C, C_2, and C_1

For changing the area (section) to map variables, specify the section type and section name through **#pragma section**.

#pragma section <section type> <section name>

Variable declaration/definition

```
#pragma section
```

When a section type is specified, only section names of the specified type can be changed.

Note that in the RX family C/C++ compiler, the section to map a variable depends on the alignment value of the variable. Examples:

B: Variables without initial values and an alignment value of four bytes are mapped

B_2: Variables without initial values and an alignment value of two bytes are mapped

B_1: Variables without initial values and an alignment value of one byte are mapped

For variables with initial values, the initial value is mapped to ROM and the variable itself is mapped to RAM (both ROM and RAM areas are necessary). When the **resetprg.c** file of the startup routine is used, the **INITSCT** function copies the initial values in ROM to the variables in RAM.

The relationship between the section type and the created section is shown in the following.

Name	Section Name	Attribute	Format Type	Initial Value and Write Oper- ation	Alignment Value
Constant area	C*1*2	romdata	Relative	Has initial values and writing is not possible	4 bytes
	C_2*1*2	romdata	Relative	Has initial values and writing is not possible	2 bytes
	C_1* ^{1*2}	romdata	Relative	Has initial values and writing is not possible	1 byte
Initialized data area	D* ^{1*2}	romdata	Relative	Has initial values and writing is possible	4 bytes
	D_2*1*2	romdata	Relative	Has initial values and writing is possible	2 bytes
	D_1* ^{1*2}	romdata	Relative	Has initial values and writing is possible	1 byte
Uninitialized data area	B* ^{1*2}	data	Relative	Does not have initial values and writing is possible	4 bytes
	B_2* ^{1*2}	data	Relative	Does not have initial values and writing is possible	2 bytes
	B_1* ^{1*2}	data	Relative	Does not have initial values and writing is possible	1 byte



Name	Section Name	Attribute	Format Type	Initial Value and Write Oper- ation	Alignment Value
switch statement branch table area	W* ¹ * ²	romdata	Relative	Has initial values and writing is not possible	4 bytes
	W_2* ^{1*2}	romdata	Relative	Has initial values and writing is not possible	2 bytes
	W_1* ^{1*2}	romdata	Relative	Has initial values and writing is not possible	1 byte
C++ initial process- ing/ postprocessing data area	C\$INT	romdata	Relative	Has initial values and writing is not possible	4 bytes
C++ virtual function table area	C\$VTBL	romdata	Relative	Has initial values and writing is not possible	4 bytes
Absolute address variable area	\$ADDR_ <section>_ <address>*³</address></section>	data	Absolute	Has or does not have initial values and writing is possible or not possible* ⁴	
Variable vector area	C\$VECT	romdata	Relative	Does not have initial values and writing is possible	_

Examples 1. Section names can be switched by the **section** option or the **#pragma section** extension. However, partial data (e.g., string literal) is not affected by **#pragma section**. For details, see the detailed description of 3.2 Extended Language Specifications**#pragma section**.

- Examples 2. Specifying a section with an alignment value of 4 when switching the section names also changes the section name of sections with an alignment value of 1 or 2. When **#pragma endian** is used to specify an endian that differs from the setting by the **endian** option, a dedicated section is created and the relevant data stored. For this section, after the section name, **_B** is added for **#pragma endian big** and **_L** is added for **#pragma endian little**. However, partial data (e.g., string literal) is not affected by **#pragma endian**. For details, see the detailed description of 3.2 Extended Language Specifications **#pragma endian**.
- Examples 3. <section> is a C, D, or B section name, and <address> is an absolute address (hexadecimal).
- Examples 4. The initial value and write operation depend on the attribute of <section>.

2.1.2 Defining Variables Used at Normal Processing and Interrupt Processing

Variables used for both normal processing and interrupt processing must be volatile qualified.

When a variable is qualified with the **volatile** qualifier, that variable is not to be optimized and optimization, such as assigning it to a register, is not performed. When operating a variable that has been **volatile** qualified, a code that reads its value from memory and writes its value to memory after operation must be used. A variable not **volatile** qualified is assigned to a register by optimization, and the code that loads that variable from memory may be deleted. When the same value is to be assigned to a variable that is not **volatile** qualified, the processing may be interpreted as redundant and the code deleted by optimization.

2.1.3 Generating a Code that Accesses Variables in the Declared Size

When accessing a variable in its declared size, the __evenaccess extended function should be used.

The <u>evenaccess</u> declaration guarantees access in the size of the variable type. The guaranteed size is a scalara type (signed char, unsigned char, signed short, unsigned short, signed int, unsigned int, signed long, or unsigned long) of four bytes or less.

The __evenaccess is invalid to the case of accessing of members by a lump of these structure and union frame.

When a structure or union is specified, the ___evenaccess declaration is effective for all members. In such a case, the access size of a scalara type member of four bytes or less is guaranteed but the access size for the whole structure or union is not guaranteed.



[Example] C source code

```
#pragma address A=0xff0178
unsigned long __evenaccess A;
void test(void)
{
     A &= ~0x20;
}
```

Output code (when evenaccess is not specified)

```
test:
    MOV.L #16712056,R1
    BCLR #5,[R1] ; 1-byte memory access
    RTS
```

Output code (when evenaccess is specified)

```
test:
    MOV.L #16712056,R1
    MOV.L [R1],R5 ; 4-byte memory access
    BCLR #5,R5
    MOV.L R5,[R1] ; 4-byte memory access
    RTS
```

2.1.4 Performing const Declaration for Variables with Unchangeable Initialized Data

A variable with an initial value is normally transferred from a ROM area to a RAM area at startup, and processing is performed using the RAM area. Accordingly, if the value is initialized data which is unchangeable in the program, the allocated RAM area goes to waste. If the const operator is added to initialized data, transfer to the RAM area at startup is disabled and the amount of used memory can be saved.

In addition, writing a program based on the rule of not changing the initial values also makes usage of ROM easier. [Example before improvement]

char a[] = { 1, 2, 3, 4, 5 };

Initial values are transferred from ROM to RAM and then processing is performed.

[Example after improvement]

const char a[] = { 1, 2, 3, 4, 5 };

Processing is performed using the initial values in ROM.

2.1.5 Defining the const Constant Pointer

The pointer is interpreted differently according to where "const" is specified.

[Example 1]

const char *p:

In this example, the object (*p) indicated by the pointer cannot be changed. The pointer itself (p) can be changed. Therefore, the result becomes as shown below and the pointer itself is mapped to RAM (section B).

p = 0; / Error */

p = 0; /* Correct */

[Example 2]

char *const p;

In this example, the pointer itself (**p**) cannot be changed. The object (***p**) indicated by the pointer can be changed. Therefore, the result becomes as shown below and the pointer itself is mapped to ROM (section C).

p = 0; / Correct */ p = 0; /* Error */

[Example 3]

char *const p:

In this example, the pointer itself (p) and the object (*p) indicated by the pointer cannot be changed. Therefore, the result becomes as shown below and the pointer itself is mapped to ROM (section C).

p = 0; / Error */

p = 0; /* Error */



2.1.6 Referencing Addresses of a Section

The start address, (end + 1) address, and size of a section can be referenced by using section address operators. _____sectop ("<section name>"): References the start address of <section name>

__secend ("<section name>"): References the (end + 1) address of <section name> secsize ("<section name>"): Generates the size of <section name>

```
[Example]
```

```
#pragma section $DSEC
static const struct {
   void *rom_s; /* Acquires the start address value of the initialized data section in
ROM */
   void *rom_e; /* Acquires the last address value of the initialized data section in
ROM */
   void *ram_s; /* Acquires the start address value of the initialized data section in
RAM */
} DTBL[]={__sectop("D"), __secend("D"), __sectop("R")};
```

The **INITSCT** function in the **resetprg.c** file of the startup routine executes transfer from ROM to RAM and initialization of uninitialized areas. The addresses acquired by **___sectop** and **___secend** written in the **dbsct.c** file are referenced during execution.

2.2 Functions

This section describes functions.

2.2.1 Filling Assembler Instructions

In the RX family C/C++ compiler, assembler instructions can be written in a C-language source program using **#pragma** inline_asm. [Example]

```
#pragma inline_asm func
static int func(int a, int b){
    ADD R2,R1 ; Assembly-language description
}
main(int *p){
    *p = func(10,20);
}
```

Inline expansion is performed for an assembly-language function specified by **#pragma inline_asm**. The general function calling rules are also applied to the calls of assembly-language inline functions.

2.2.2 Performing In-Line Expansion of Functions

#pragma inline declares a function for which inline expansion is performed.

The compiler options **inline** and **noline** are also used to enable or disable inline expansion. However, even when the **noinline** option is specified, inline expansion is done for the function specified by **#pragma inline**.

A global function or a static function member can be specified as a function name. A function specified by **#pragma** inline or a function with specifier inline (C++ and C (C99)) are expanded where the function is called.



[Example] <u>C source code</u>

```
#pragma inline(func)
static int func (int a, int b)
{
    return (a+b)/2;
}
int x;
main()
{
    x=func(10,20);
}
```

Expanded image

```
int x;
main()
{
    int func_result;
        {
            int a_1=10, b_1=20;
            func_result=(a_1+b_1)/2;
        }
        x=func_result;
}
```

2.2.3 Performing (Inter-File) In-Line Expansion of Functions

Normally, inline expansion is performed for functions within a file. However, using the **-file_inline** option of the compiler allows inline expansion to be performed for even inter-file function calling. [Example]

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

By compiling with the specification of ccrx -inline -file_inline=b.c a.c, calling of function g in a.c is expanded and becomes as follows:

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

2.3 Using Microcomputer Functions

This section describes usage of microcomputer functions.

2.3.1 Processing an Interrupt in C Language

```
Use #pragma interrupt to declare an interrupt function.
[Example]
<u>C source code</u>
```

```
#pragma interrupt func
void func(){ .... }
```



Generated code

2.3.2 Using CPU Instructions in C Language

The compiler provides the following intrinsic functions for cases of accessing control registers and special instructions that cannot be expressed in C language.

- 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
- The intrinsic functions are listed below.

Specifications	Function
signed long max(signed long data1, signed long data2)	Selects the maximum value.
signed long min(signed long data1, signed long data2)	Selects the minimum value.
unsigned long revl(unsigned long data)	Reverses the byte order in longword data.
unsigned long revw(unsigned long data)	Reverses the byte order in longword data in word units.
void xchg(signed long *data1, signed long *data2)	Exchanges data.
long long rmpab(long long init, unsigned long count, signed char *addr1, signed char *addr2)	Multiply-and-accumulate operation (byte).
long long rmpaw(long long init, unsigned long count, short *addr1, short *addr2)	Multiply-and-accumulate operation (word).
long long rmpal(long long init, unsigned long count, long *addr1, long *addr2)	Multiply-and-accumulate operation (long- word).
unsigned long rolc(unsigned long data)	Rotates data including the carry to left by one bit.
unsigned long rorc(unsigned long data)	Rotates data including the carry to right by one bit.
unsigned long rotl(unsigned long data, unsigned long num)	Rotates data to left.
unsigned long rotr(unsigned long data, unsigned long num)	Rotates data to right.
void brk(void)	BRK instruction exception.
void int_exception(signed long num)	INT instruction exception.
void wait(void)	Stops program execution.
void nop(void)	Expanded to a NOP instruction.



Specifications	Function	
void set_ipl(signed long level)	Sets the interrupt priority level.	
unsigned char get_ipl(void)	Refers to the interrupt priority level.	
void set_psw(unsigned long data)	Sets a value for PSW .	
unsigned long get_psw(void)	Refers to the PSW value.	
void set_fpsw(unsigned long data)	Sets a value for FPSW .	
unsigned long get_fpsw(void)	Refers to the FPSW value.	
void set_usp(void *data)	Sets a value for USP .	
void *get_usp(void)	Refers to the USP value.	
void set_isp(void *data)	Sets a value for ISP .	
void *get_isp(void)	Refers to the ISP value.	
void set_intb(void *data)	Sets a value for INTB .	
void *get_intb(void)	Refers to the INTB value.	
void set_bpsw(unsigned long data)	Sets a value for BPSW .	
unsigned long get_bpsw(void)	Refers to the BPSW value.	
void set_bpc(void *data)	Sets a value for BPC .	
void *get_bpc(void)	Refers to the BPC value.	
void set_fintv(void *data)	Sets a value for FINTV .	
void *get_fintv(void)	Refers to the FINTV value.	
unsigned long long emulu(unsigned long, unsigned long)	Unsigned multiplication of valid 64 bits	
signed long long emul(signed long, signed long)	Signed multiplication of valid 64 bits	

2.4 Variables (Assembly Language)

This section describes variables (assembly language).

2.4.1 Defining Variables without Initial Values

Allocate a memory area in a **DATA** section.

To define a **DATA** section, use the **.SECTION** directive. To allocate a memory area, use the **.BLKB** directive for specification in 1-byte units, the **.BLKW** directive for 2-byte units, the **.BLKL** directive for 4-byte units, and the **.BLKD** directive for 8-byte units.

[Example]

.SECTION area,DATA work1:.BLKB 1; Allocates a RAM area in 1-byte units work2:.BLKW 1; Allocates a RAM area in 2-byte units work3:.BLKL 1; Allocates a RAM area in 4-byte units work4:.BLKD 1; Allocates a RAM area in 8-byte units

2.4.2 Defining a cost Constant with an Initial Value

Initialize a memory area in a **ROMDATA** section.



To define a **ROMDATA** section, use the **.SECTION** directive. To initialize memory, use the **.BYTE** directive for 1 byte, the **.WORD** directive for 2 bytes, the **.LWORD** directive for 4 bytes, the **.FLOAT** directive for floating-point 4 bytes, and the **.DOUBLE** directive for floating-point 8 bytes.

[Example]

.SECTION value,ROMDATA				
work1:.BYTE "data";	Stores 1-byte fixed data in ROM			
work2:.WORD "data";	Stores 2-byte fixed data in ROM			
work3:.LWORD "data";	Stores 4-byte fixed data in ROM			
work4:.FLOAT 5E2;	Stores 4-byte floating-point data in ROM			
work5:.DOUBLE 5E2;	Stores 8-byte floating-point data in ROM			

2.4.3 Referencing the Address of a Section

The size and start address of a section that were specified as operands using the **SIZEOF** and **TOPOF** operators are handled as values.

[Example]

```
...
MVTC #(TOPOF SU + SIZEOF SU),USP
; Sets the user stack area address to USP as (SU start address + SU size)
MVTC #(TOPOF SI + SIZEOF SI),ISP
; Sets the interrupt stack area address to ISP as (SI start address + SI size)
...
```

2.5 Startup Routine

This section describes the startup routine.

2.5.1 Allocating Stack Areas

Since the **PowerON_Reset_PC** function in the **resetprg.c** file of the startup routine is declared by **"#pragma entry**", the compiler and optimizing linkage editor automatically generate the initialization code for the user stack **USP** and interrupt stack **ISP** at the top of the function, based on the settings below.

- (1) Setting the User Stack Specify the size of the stack area by **#pragma stacksize su=0xXXX** in the **stacksct.h** file, and specify the loca
 - tion of the SU section by the -start option of the optimizing linkage editor.
- (2) Setting the Interrupt Stack

Specify the size of the stack area by **#pragma stacksize si=0xXXX** in the **stacksct.h** file, and specify the location of the **SI** section by the **-start** option of the optimizing linkage editor. [Example]

```
<resetprg.c>
...
#pragma section ResetPRG
#pragma entry PowerON_Reset_PC
void PowerON_Reset_PC(void)
{
...
<stacksct.h>
#pragma stacksize su=0x300
#pragma stacksize si=0x100
```

[Generated code example]

. . .



2.5.2 Initializing RAM

The _INITSCT function in the resetprg.c file of the startup routine is used to initialize uninitialized areas. To add a section to be initialized, add the following description to the dbsct.c file.

```
[Example]
 <dbsct.c>
 . . .
 #pragma section C C$BSEC
 extern const struct {
     _UBYTE *b_s;
                           /* Start address of non-initialized data section */
                         /* End address of non-initialized data section */
     UBYTE *b e;
 }
     \_BTBL[] = {
      { __sectop("B"), __secend("B") },
      { __sectop("B_2"), __secend("B_2") },
      { __sectop("B_1"), __secend("B_1") }
 };
 . . .
```

In the above example, the addresses used in the **INITSCT** function are stored in the table in order to initialize the **B**, **B_2**, and **B_1** sections.

2.5.3 Transferring Variables with Initial Values from ROM to RAM

The _INITSCT function in the resetprg.c file of the startup routine is used to transfer variables with initial values from ROM to RAM. To add a section to be transferred, add the following description to the dbsct.c file. [Example]

```
<dbsct.c>
. . .
#pragma section C C$DSEC
extern const struct {
                         /* Start address of the initialized data section in ROM */
    _UBYTE *rom_s;
                        /* End address of the initialized data section in ROM */
    _UBYTE *rom_e;
    _UBYTE *ram_s;
                         /* Start address of the initialized data section in RAM */
}
    DTBL[] = {
     { __sectop("D"), __secend("D"), __sectop("R") },
     { __sectop("D_2"), __secend("D_2"), __sectop("R_2") },
     { __sectop("D_1"), __secend("D_1"), __sectop("R_1") }
};
. . .
```

In the above example, the addresses used in the **INITSCT** function are stored in the table in order to transfer the contents of the D, D_2, and D_1 sections to the R, R_2, and R_1 sections. Note that the location addresses of the D, D_2, D_1, R, R_2, and R_1 sections should be specified by the

-start option of the optimizing linkage editor. The relocation solution by transferring data from ROM to RAM should be specified by the -rom option of the optimizing linkage editor.

2.6 Reducing the Code Size

This section describes code size reduction.

2.6.1 Data Structure

In a case where related data is referenced many times in the same function, usage of a structure will facilitate generation of a code using relative access, and an improvement in efficiency can be expected. The efficiency will also be improved when data is passed as arguments. Because the access range of relative access is limited, it is effective to place the frequently accessed data at the top of the structure.

When data takes the form of a structure, it is easy to perform tuning that changes the data expressions. [Example]

Numeric values are assigned to variables $\boldsymbol{a},\,\boldsymbol{b},$ and $\boldsymbol{c}.$



Source code before improvement

Assembly-language expansion code before improvement

```
_func:

MOV.L #_a,R4

MOV.L #00000001H,[R4]

MOV.L #_b,R4

MOV.L #00000002H,[R4]

MOV.L #_c,R4

MOV.L #0000003H,[R4]

RTS
```

Source code after improvement

```
struct s{
    int a;
    int b;
    int c;
} s1;
void func()
{
    register struct s *p=&s1;
    p->a = 1;
    p->b = 2;
    p->c = 3;
}
```

Assembly-language expansion code after improvement

```
_func:

MOV.L #_s1,R5

MOV.L #00000001H,[R5]

MOV.L #0000002H,04H[R5]

MOV.L #0000003H,08H[R5]

RTS
```

2.6.2 Local Variables and Global Variables

Variables that can be used as local variables must be declared as local variables and not as global variables. There is a possibility that the value of a global variable will be changed by function calling or pointer operations, thus the efficiency of optimization is degraded.

The following advantages are available when local variables are used.

- Access cost is low
- May be assigned to a register
- Efficiency of optimization is good



[Example]

Case in which global variables are used for temporary variables (before improvement) and case in which local variables are used (after improvement)

Source code before improvement

```
int tmp;
void func(int* a, int* b)
{
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Assembly-language expansion code before improvement

```
__func:

MOV.L #_tmp,R4

MOV.L [R1],[R4]

MOV.L [R2],[R1]

MOV.L [R4],[R2]

RTS
```

Source code after improvement

```
void func(int* a, int* b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Assembly-language expansion code after improvement

```
__func:
MOV.L [R1],R5
MOV.L [R2],[R1]
MOV.L R5,[R2]
RTS
```

2.6.3 Offset for Structure Members

A structure member is accessed after adding the offset to the structure address. Since a small offset is advantageous for the size, members often used should be declared at the top.

The most effective combination is within 32 bytes from the top for the **signed char** or **unsigned char** type, within 64 bytes from the top for the **short** or **unsigned short** type, or within 128 bytes from the top for the **int**, **unsigned int**, **long**, or **unsigned long** type.

```
[Example]
```

An example in which the code is changed because of the offset of the structure is shown below. <u>Source code before improvement</u>

```
struct str {
    long L1[8];
    char C1;
};
struct str STR1;
char x;
void func()
{
    x = STR1.C1;
}
```



Assembly-language expansion code before improvement

_func:

```
MOV.L #_STR1,R4
MOVU.B 20H[R4],R5
MOV.L #_x,R4
MOV.B R5,[R4]
RTS
```

Source code after improvement

```
struct str {
     char C1;
     long L1[8];
};
struct str STR1;
char x;
void func()
{
     x = STR1.C1;
}
```

Assembly-language expansion code after improvement

```
func:
    MOV.L #_STR1,R4
    MOVU.B [R4],R5
    MOV.L #_x,R4
    MOV.B R5,[R4]
    RTS
```

Note

When defining a structure, declare the members while considering the boundary alignment value. The boundary alignment value of a structure is the most largest boundary alignment value within the structure. The size of a structure becomes a multiple of the boundary alignment value. For this reason, when the end of a structure does not match the boundary alignment value of the structure itself, the size of the structure also includes the unused area that was created for guaranteeing the next boundary alignment. Source code before improvement

```
/* Boundary alignment value is 4 because the maximum member is the int type */
struct str {
     char C1; /* 1 byte + 3 bytes of boundary alignment */
     long L1; /* 4 bytes */
    char C2; /* 1 byte */
    char C3; /* 1 byte */
```

char C4; /* 1 byte + 1 byte of boundary alignment */

}STR1;

str size before improvement

```
.SECTION B, DATA, ALIGN=4
    .glb _STR1
_STR1: ; static: STR1
     .blkl 3
```



Source code after improvement

```
/* Boundary alignment value is 4 because the maximum member is the int type */
struct str {
    char C1; /* 1 byte */
    char C2; /* 1 byte */
    char C3; /* 1 byte */
    char C4; /* 1 byte */
    long L1; /* 4 bytes */
}STR1;
```

str size after improvement

```
.SECTION B,DATA,ALIGN=4
.glb _STR1
_STR1: ; static: STR1
.blkl 2
```

2.6.4 Allocating Bit Fields

To set members of different bit fields, the data including the bit field needs to be accessed each time. These accesses can be kept down to one access by collectively allocating the related bit fields to the same structure. [Example]

An example in which the size is improved by allocating bit fields related to the same structure is shown below. Source code before improvement

```
struct str
{
    Int flag1:1;
}b1,b2,b3;
void func()
{
    b1.flag1 = 1;
    b2.flag1 = 1;
    b3.flag1 = 1;
}
```

Assembly-language expansion code before improvement

```
_func:

MOV.L #_b1,R5

BSET #00H,[R5]

MOV.L #_b2,R5

BSET #00H,[R5]

MOV.L #_b3,R5

BSET #00H,[R5]

RTS
```

Source code after improvement

```
struct str
{
    int flag1:1;
    int flag2:1;
    int flag3:1;
}al;
void func()
{
    al.flag1 = 1;
    al.flag2 = 1;
    al.flag3 = 1;
}
```



Assembly-language expansion code after improvement

```
_func:
MOV.L #_al,R4
MOVU.B [R4],R5
OR #07H,R5
MOV.B R5,[R4]
RTS
```

2.6.5 Optimization of External Variable Accesses when the Base Register is Specified

When **R13** is specified as the base register of the RAM section, accesses to the RAM section are performed relative to the **R13** register. Furthermore, if optimization of inter-module external variable accesses is enabled, the value relative to the **R13** register is optimized, and the instruction size becomes smaller if the value is 8 bits or less.

```
[Example]
```

Source code before improvement

Assembly-language expansion code before improvement

```
_func:

MOV.L #_a,R4

MOV.L #0000000H,[R4]

MOV.L #_b,R4

MOV.L #00000001H,{R4}

MOV.L #_c,R4

MOV.L #0000002H,[R4]

MOV.L #_d,[R4]

MOV.L #0000003H,[R4]

RTS
```

Source code after improvement

Assembly-language expansion code after improvement

```
_func:

MOV.L #000000H,_a-__RAM_TOP:16[R13]

MOV.L #0000001H,_b-__RAM_TOP:16[R13]

MOV.L #000002H,_c-__RAM_TOP:16[R13]

MOV.L #000003H,_d-__RAM_TOP:16[R13]

RTS
```



2.6.6 Specified Order of Section Addresses by Optimizing Linkage Editor at Optimization of External Variable Accesses

In an instruction that accesses memory in the register relative-address format, the instruction size is small when the displacement value is small.

In some cases, the code size can be improved when the order of allocating the sections by the optimizing linkage editor is changed with reference to the following guidelines.

- Place at the beginning the sections of external variables that are frequently accessed in the function.

- Place at the beginning the sections of external variables with small type sizes.

Note however that the build time gets longer when external variable accesses are optimized because the compiler runs twice.

[Example] Source code before improvement

```
/* Section D_1 */
char d11=0, d12=0, d13=0, d14=0;
/* Section D_2 */
short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};
/* Section D */
int d41=0, d42=0, d43=0, d44=0, dmy4[60]=\{0\}
void func(int a){
    d11 = a;
    d12 = a;
    d13 = a;
    d14 = a;
    d21 = a;
    d22 = a;
    d23 = a;
    d24 = a;
    d41 = a;
    d42 = a;
    d43 = a;
    d44 = a;
}
```

Assembly-language expansion code before improvement

```
<When the allocation order of sections is "D, D_2, D_1" or D*>
func:
    MOV.L #d41,R4
    MOV.B R1,0120H[R4]
    MOV.B R1,0121H[R4]
    MOV.B R1,0122H[R4]
    MOV.B R1,0123H[R4]
    MOV.W R1,0100H[R4]
    MOV.W R1,0102H[R4]
    MOV.W R1,0104H[R4]
    MOV.W R1,0106H[R4]
    MOV.L R1,[R4]
    MOV.L R1,04H[R4]
    MOV.L R1,08H[R4]
    MOV.L R1,0CH[R4]
    RTS
```



Source code after improvement

```
/* Section D_1 */
char d11=0, d12=0, d13=0, d14=0;
 /* Section D_2 */
short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};
 /* Section D */
 int d41=0, d42=0, d43=0, d44=0, dmy4[60]=\{0\}
 void func(int a){
      d11 = a;
      d12 = a;
      d13 = a;
      d14 = a_i
      d21 = a;
      d22 = a;
      d23 = a;
      d24 = a;
      d41 = a;
      d42 = a;
      d43 = a;
      d44 = a;
 }
Assembly-language expansion code after improvement
 <When the allocation order of sections is "D_1, D_2, D" or D*>
 _func:
      MOV.L #d11,R4
      MOV.B R1,[R4]
      MOV.B R1,01H[R4]
      MOV.B R1,02H[R4]
      MOV.B R1,03H[R4]
      MOV.W R1,04H[R4]
      MOV.W R1,06H[R4]
      MOV.W R1,08H[R4]
      MOV.W R1,0AH[R4]
```

MOV.L R1,24H[R4] MOV.L R1,28H[R4] MOV.L R1,2CH[R4] MOV.L R1,30H[R4]

RTS

2.6.7 Modularization of Functions

When calling a function in a different file, the process is expanded into a 4-byte **BSR** instruction. However, when calling a function in the same file, the process is expanded into a 3-byte **BSR** instruction if the calling range is near, and a compact object can be generated.

Modularization also makes corrections at tune-up to be performed easily.

```
[Example]
Function g is called from function f.
Source code before improvement
extern void sub(void);
int func()
{
    sub();
```

return(0);
}



Assembly-language expansion code before improvement

```
_func:
BSR _sub ;length A
MOV.L #00000000H,R1
RTS
```

Source code after improvement

```
void sub(void);
int func()
{
     sub();
     return(0);
}
```

Assembly-language expansion code after improvement

```
_func:
BSR _sub;length W
MOV.L #00000000H,R1
RTS
```

2.6.8 Interrupt

Due to many registers being saved and restored before and after an interrupt processing, the expected interrupt response time may not be obtained. In such a case, the fast interrupt setting (fint) and fint_register option should be used to keep down the number of saving and restoring of registers so that the interrupt response time can be reduced.

Note however that usage of the **fint_register** option limits the usable registers in other functions so the efficiency of the entire program is degraded in some cases. [Example]

Source code before improvement

```
#pragma interrupt int_func
volatile int count;
void int_func()
{
     count++;
}
```

Assembly-language expansion code before improvement

```
_int_func:

PUSHM R4-R5

MOV.L #_count,R4

MOV.L [R4],R5

ADD #01H,R5

MOV.L R5,[R4]

POPM R4-R5

RTE
```

Source code after improvement

```
#pragma interrupt int_func(fint)
volatile int count;
void int_func()
{
     count++;
}
```



Assembly-language expansion code after improvement

```
<When the fint_register=2 option is specified>
_int_func:
    MOV.L #_count,R12
    MOV.L [R12],[R13]
    ADD #01H,R13
    MOV.L R13,[R12]
    RTFI
```

2.7 High-Speed Processing

This section describes high-speed processing.

2.7.1 Loop Control Variable

Loop expansion cannot be optimized if there is a possibility that the size difference prevents the loop control variable from expressing the data to be compared when determining whether the loop end condition is met. For example, if the loop control variable is **signed char** while the data to be compared is **signed long**, loop expansion is not optimized. Thus, compared to **signed char** and **signed short**, it is easier to perform optimization of loop expansion for **signed long**. To optimize loop expansion, specify the loop control variable as a 4-byte integer type.

[Example] Source code before improvement

```
signed long array_size=16;
signed char array[16];
void func()
{
    signed char I;
    for(i=0;i<array_size;i++)
    {
        array[i]=0;
    }
}
```

Assembly-language expansion code before improvement

```
<When loop=2 is specified>
_func:
     MOV.L #_array_size,R4
     MOV.L [R4],R2
     MOV.L #0000000H,R5
     BRA L11
L12:
     MOV.L #_array,R14
     MOV.L #0000000H,R3
     MOV.B R3,[R5,R4]
     ADD #01H,R5
L11:
     MOV.B R5,R5
     CMP R2,R5
     BLT L12
L13:
     RTS
```



Source code after improvement

```
signed long array_size=16;
signed char array[16];
void func()
{
    signed long I;
    for(i=0;i<array_size;i++)
    {
        array[i]=0;
    }
}
```

Assembly-language expansion code after improvement

```
<When loop=2 is specified>
_func:
     MOV.L #_array_size,R5
     MOV.L [R5],R2
     MOV.L #0000000H,R4
     ADD #0FFFFFFFH,R2,R3
     CMP R3,R2
     BLE L12
L11:
     MOV.L #_array,R1
     MOV.L R1,R5
     BRA L13
L14:
     MOV.W #0000H,[R5]
     ADD #02H,R5
     ADD #02H,R4
L13:
     CMP R3,R4
     BLT L14
L15:
     CMP R2,R4
     BGE L17 L16:
     MOV.L #0000000H,R5
     MOV.B R5,[R4,R1]
     RTS
L12:
     MOV.L #_array,R5
     MOV.L #0000000H,R3
L19:
     CMPR2,R4
     BGE L17
L20:
     MOV.B R3,[R5+]
     ADD #01H,R4
     BRA L19
L17:
     RTS
```

2.7.2 Function Interface

The number of arguments should be carefully selected so that all arguments can be set in registers (up to four). If there are too many arguments, turn them into a structure and pass the pointer. If the structure itself is passed through and forth, instead of the pointer of the structure, the structure may be too large to be set in a register. When arguments are set in registers, calling and processing at the entry and exit of the function can be simplified. In addition, space in the stack area can be saved. Note that registers **R1** to **R4** are to be used for arguments.

[Example]



Function **f** has four more arguments than the number of registers for arguments. <u>Source code before improvement</u>

```
void call_func()
{
func(1,2,3,4,5,6,7,8);
}
```

Assembly-language expansion code before improvement

```
_call_func:

SUB #04H,R0

MOV.L #08070605H,[R0]

MOV.L #00000004H,R4

MOV.L #00000003H,R3

MOV.L #00000002H,R2

MOV.L #00000001H,R1

BSR _func

ADD #04H,R0

RTS
```

Source code after improvement

```
struct str{
        char a;
        char b;
        char c;
        char d;
        char d;
        char f;
        char g;
        char h;
};
struct str arg = {1,2,3,4,5,6,7,8};
void call_func()
{
        func(&arg);
}
```

Assembly-language expansion code after improvement

```
_call_func:
MOV.L #arg,R1
BRA _func
```

2.7.3 Reducing the Number of Loops

Loop expansion is especially effective for inner loops. Since the program size is increased by loop expansion, loop expansion should be performed when a fast execution speed is preferred at the expense of the program size. [Example]

```
Array a[] is initialized.
Source code before improvement
```

```
extern int a[100];
void func()
{
    int I;
    for( i = 0 ; i < 100 ; i++ ){
        a[i] = 0;
    }
}</pre>
```



Assembly-language expansion code before improvement

```
_func:

    MOV.L #00000064H,R4

    MOV.L #_a,R5

    MOV.L #00000000H,R3

L11:

    MOV.L R3,[R5+]

    SUB #01H,R4

    BNE L11

L12:

    RTS
```

Source code after improvement

```
extern int a[100];
void func()
{
    int I;
    for( i = 0 ; i < 100 ; i+=2 )
    {
        a[i] = 0;
        a[i+1] = 0;
    }
}</pre>
```

Assembly-language expansion code after improvement

```
_func:

    MOV.L #0000032H,R4

    MOV.L #_a,R5

L11:

    MOV.L #00000000H,[R5]

    MOV.L #0000000H,04H[R5]

    ADD #08H,R5

    SUB #01H,R4

    BNE L11

L12:

    RTS
```

2.7.4 Usage of a Table

If the processing in each **case** label of a **switch** statement is almost the same, consider the usage of a table. [Example]

The character constant to be assigned to variable **ch** is changed by the value of variable **i**. <u>Source code before improvement</u>

```
char func(int i)
{
    char ch;
    switch (i) {
        case 0:
            ch = `a'; break;
        case 1:
            ch = `x'; break;
        case 2:
            ch = `b'; break;
    }
    return(ch);
}
```



Assembly-language expansion code before improvement

```
_func:
     CMP #00H,R1
     BEQ L17
L16:
     CMP #01H,R1
     BEQ L19
     CMP #02H,R1
     BEQ L20
     BRA L21
L12:
L17:
     MOV.L #00000061H,R1
     BRA L21
L13:
L19:
     MOV.L #00000078H,R1
     BRA L21
L14:
L20:
     MOV.L #00000062H,R1
L11:
L21:
     MOVU.B R1,R1
     RTS
```

Source code after improvement

char chbuf[] = {'a', 'x', 'b'};
char func(int i)
{
 return (chbuf[i]);
}

Assembly-language expansion code after improvement

```
MOV.L #_chbuf,R4
MOVU.B [R1,R4],R1
RTS
```

2.7.5 Branch

_f

When comparison is performed in order beginning at the top, such as in an **else if** statement, the execution speed in the cases at the end gets slow if there is many branching. Cases with frequent branching should be placed near the beginning.


[Example]

The return value changes depending on the value of the argument. Source code before improvement

```
int func(int a)
{
    if (a==1)
        a = 2;
    else if (a==2)
        a = 4;
    else if (a==3)
        a = 0;
    else
        a = 0;
    return(a);
}
```

Assembly-language expansion code before improvement

```
_func:
    CMP #01H,R1
    BEQ L11
L12:
    CMP #02H,R1
    BNE L14
L13:
    MOV.L #0000004H,R1
    RTS
L14:
    CMP #03,R1
    BNE L17
L16:
    MOV.L #0000008H,R1
    RTS
L17:
    MOV.L #0000000H,R1
    RTS
L11:
    MOV.L #0000002H,R1
    RTS
```

Source code after improvement

```
int func(int a)
{
    if (a==3)
        a = 8;
    else if (a==2)
        a = 4;
    else if (a==1)
        a = 2;
    else
        a = 0;
    return (a);
}
```



Assembly-language expansion code after improvement

```
_func:
     CMP #03H,R1
     BEQ L11
L12:
     CMP #02H,R1
     BNE L14
L13:
     MOV.L #0000004H,R1
     RTS
L14:
     CMP #01H,R1
     NE L17
L16:
     MOV.L #0000002H,R1
     RTS
L17:
     MOV.L #0000000H,R1
     RTS
L11:
     MOV.L #0000008H,R1
     RTS
```

2.7.6 Inline Expansion

The execution speed can be improved by performing inline expansion for functions that are frequently called. A significant effect may be obtained by expanding functions that are particularly called in the loop. However, since the program size is inclined to be increased by inline expansion, inline expansion should be performed when a fast execution speed is preferred at the expense of the program size.

[Example]

The elements of array **a** and array **b** are exchanged. Source code before improvement

```
int x[10], y[10];
static void sub(int *a, int *b, int I)
{
     int temp;
     temp = a[i];
     a[i] = b[i];
     b[i] = temp;
}
void func()
{
     int I;
     for(i=0;i<10;i++)</pre>
     {
           sub(x,y,i);
     }
}
```



Assembly-language expansion code before improvement

```
__$sub:
     SHLL #02H,R3
     ADD R3,R1
     MOV.L [R1],R5
     ADD R3,R2
     MOV.L [R2],[R1]
     MOV.L R5,[R2]
     RTS
_func:
     PUSHM R6-R8
     MOV.L #0000000H,R6
     MOV.L #_x,R7
     MOV.L #_y,R8
L12:
     MOV.L R6,R3
     MOV.L R7,R1
     MOV.L R8,R2
     ADD #01H,R6
     BSR __$sub
     CMP #0AH,R6
     BLT L12
L13:
     RTSD #0CH,R6-R8
```

Source code after improvement

```
int x[10], y[10];
#pragma inline(sub)
static void sub(int *a, int *b, int I)
{
     int temp;
     temp = a[i];
     a[i] = b[i];
     b[i] = temp;
}
void func()
{
     int I;
     for(i=0;i<10;i++)</pre>
     {
          sub(x,y,i);
     }
}
```

Assembly-language expansion code after improvement



2.8 Mutual Reference between Compiler and Assembler

This section describes mutual referencing between the compiler and assembler.

External names which have been declared in a C/C++ program can be referenced and updated in both directions between the C/C++ program and an assembly-language program. The compiler treats the following items as external names.

- Global variables which are not declared as static storage classes (C/C++ programs)
- Variable names declared as extern storage classes (C/C++ programs)
- Function names not specified as static storage classes (C programs)
- Non-member, non-inline function names not specified as static storage classes (C++ programs)
- Non-inline member function names (C++ programs)
- Static data member names (C++ programs)

2.8.1 Referencing Assembly-Language Program External Names in C/C++ Programs

In assembly-language programs, **.EXPORT** is used to declare external symbol names (preceded by an underscore (_)). In C/C++ programs, symbol names (not preceded by an underscore) are declared using the **extern** keyword. [Example of assembly-language source]

```
.glb _a, _b
.SECTION D,ROMDATA,ALIGN=4
_a: .LWORD 1
_b: .LWORD 1
.END
[Example of C source]
```

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

2.8.2 Referencing C/C++ Program External Names (Variables and C Functions) from Assembly-Language Programs

A C/C++ program can define external variable names (without an underscore (_)). In an assembly-language program, **.IMPORT** is used to declare an external name (preceded by an underscore). [Example of C source]

int a;

[Example of assembly-language source]

```
.GLB _a
.SECTION P,CODE
MOV.L #A_a,R1
MOV.L [R1],R2
ADD #1,R2
MOV.L R2,[R1]
RTS
.SECTION D,ROMDATA,ALIGN=4
A_a: .LWORD _a
.END
```



2.8.3 Referencing C++ Program External Names (Functions) from Assembly-Language Programs

By declaring functions to be referenced from an assembly-language program using the **extern "C"** keyword, the function can be referenced using the same rules as in (2) above. However, functions declared using **extern "C"** cannot be overloaded.

```
[Example of C++ source]
```

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

[Example of assembly-language source]

```
.GLB _sub
.SECTION P,CODE
:
PUSH.L R13
MOV.L 4[R0],R1
MOV.L R3,R12
MOV.L #_sub,R14
JSR R14
POP R13
RTS
:
.END
```



3. Compiler Language Specifications

3.1 Basic Language Specifications

The RXC supports the language specifications stipulated by the ANSI standards. These specifications include items that are stipulated as processing definitions. This chapter explains the language specifications of the items dependent on the processing system of the RX microcontrollers.

For extended language specifications explicitly added by the RXC, refer to section 3.2 Extended Language Specifications.

3.1.1 Unspecified Behavior

This section describes behavior that is not specified by the ANSI standard.

- (1) Execution environment initialization of static storage Static data is output during compilation as a data section.
- (2) Meanings of character displays backspace (\b), horizontal tab (\t), vertical tab (\t) This is dependent on the design of the display device.
- (3) Types floating point Conforms to IEEE754*.

Note IEEE: Institute of Electrical and Electronics Engineers IEEE754 is a system for handling floating-point calculations, providing a uniform standard for data formats, numerical ranges, and the like handled.

- (4) Expressions evaluation order In general, expressions are evaluated from left to right. The behavior when optimization has been applied, however, is undefined. Options or other settings could change the order of evaluation, so please do not code expressions with side effects.
- (5) Function calls parameter evaluation order In general, function arguments are evaluated from first to last. The behavior when optimization has been applied, however, is undefined. Options or other settings could change the order of evaluation, so please do not code expressions with side effects.
- (6) Structure and union specifiers These are adjusted so that they do no span bit field type alignment boundaries. If packing has been conducting using options or a #pragma, then bit fields are packed, and not adjusted to alignment boundaries.
- (7) Function definitions storage of formal parameters These are assigned to the stack and register. For details, refer to 3.4 Function Calling Interface.
- (8) # operator These are evaluated left to right.

3.1.2 Undefined Behavior

This section describes behavior that is not defined by the ANSI standard.

- Character set A message is output if a source file contains a character not specified by the character set.
- (2) Lexical elements A message is output if there is a single or double quotation mark ("/") in the last category (a delimiter or a single non-whitespace character that does not lexically match another preprocessing lexical type).
- (3) Identifiers Since all identifier characters have meaning, there are no meaningless characters.
- (4) Identifier binding A message is output if both internal and external binding was performed on the same identifier within a translation unit.
- (5) Compatible type and composite type All declarations referencing the same object or function must be compatible. Otherwise, a message will be output.

(6) Character constants

Specific non-graphical characters can be expressed by means of extended notation, consisting of a backslash (\) followed by a lower-case letter. The following are available: \a, \b, \f, \n, \r, \t, and \v. There is no other extended notation; other letters following a backslash (\) become that letter.

- (7) String literals concatenation When a simple string literal is adjacent to a wide string literal token, simple string concatenation is performed.
- (8) String literals modification Users modify string literals at their own risk. Although the string will be changed if it is allocated to RAM, it will not be changed if it is allocated to ROM.
- (9) Header names

If the following characters appear in strings between the delimiter characters < and >, or between two double quotation marks ("), then they are treated as part of the file name: characters, comma (,), double quote ("), two slashes (//), or slash-asterisk (/*). The backslash (\) is treated as a folder separator.

- (10) Floating point type and integral type If a floating-point type is converted into an integral type, and the integer portion cannot be expressed as an integral type, then the value is truncated until it can.
- (11) Ivalues and function specifiersA message is output if an incomplete type becomes an Ivalue.
- (12) Function calls number of arguments If there are too few arguments, then the values of the formal parameters will be undefined. If there are too many arguments, then the excess arguments will be ignored when the function is executed, and will have no effect. A message will be output if there is a function declaration before the function call.
- (13) Function calls types of extended parameters If a function is defined without a function prototype, and the types of the extended arguments do not match the types of the extended formal parameters, then the values of the formal parameters will be undefined.
- (14) Function calls incompatible types If a function is defined with a type that is not compatible with the type specified by the expression indicating the called function, then the return value of the function will be invalid.
- (15) Function declarations incompatible types If a function is defined in a form that includes a function prototype, and the type of an extended argument is not compatible with that of a formal parameter, or if the function prototype ends with an ellipsis, then it will be interpreted as the type of the formal parameter.
- (16) Addresses and indirection operators If an incorrect value is assigned to a pointer, then the behavior of the unary * operator will either obtain an undefined value or result in an illegal access, depending on the hardware design and the contents of the incorrect value.
- (17) Cast operator function pointer casts If a typecast pointer is used to call a function with other than the original type, then it is possible to call the function. If the parameters or return value are not compatible, then it will be invalid.
- (18) Cast operator integral type casts If a pointer is cast into an integral type, and the amount of storage is too small, then the storage of the cast type will be truncated.
- (19) Multiplicative operators
 A message will be output if a divide by zero is detected during compilation.

 During execution, a divide by zero will raise an exception. If an error-handling routine has been coded, it will be handled by this routine.
- (20) Additive operators non-array pointers If addition or subtraction is performed on a pointer that does other than indicate elements in an array object, the behavior will be as if the pointer indicates an array element.
- (21) Additive operators subtracting a pointer from another array If subtraction is performed using two pointers that do not indicate elements in the same array object, the behavior will be as if the pointers indicate array elements.
- (22) Bitwise shift operators If the value of the right operand is negative, or greater than the bit width of the extended left operand, then the result will be the shifted value of the right operand, masked by the bit width of the left operand.



(23) Function operators - pointers

If the objects referring to by the pointers being compared are not members of the same structure or union object, then the relationship operation will be performed for pointers referring to the same object.

(24) Simple assignment

If a value stored in an object is accessed via another object that overlaps that object's storage area in some way, then the overlapping portion must match exactly. Furthermore, the types of the two objects must have modified or non-modified versions with compatible types. Assignment to non-matching overlapping storage could cause the value of the assignment source to become corrupted.

(25) Structure and union specifiers

If the member declaration list does not include named members, then a message will be output warning that the list has no effect. Note, however, that the same message will be output accompanied by an error if the -Xansi option is specified.

(26) Type modifiers - const

A message will be output if an attempt is made to modify an object defined with a const modifier, using an Ivalue that is the non-const modified version. Casting is also prohibited.

(27) Type modifiers - volatile

A message will be output if an attempt is made to modify an object defined with a volatile modifier, using an lvalue that is the non-volatile modified version.

(28) return statements

A message will be output if a return statement without an expression is executed, and the caller uses the return value of the function, and there is a declaration. If there is no declaration, then the return value of the function will be undefined.

(29) Function definitions

If a function taking a variable number of arguments is defined without a parameter type list that ends with an ellipsis, then the values of the formal parameters will be undefined.

(30) Conditional inclusion

If a replacement operation generates a "defined" token, or if the usage of the "defined" unary operator before macro replacement does not match one of the two formats specified in the constraints, then it will be handled as an ordinary "defined".

- (31) Macro replacement arguments not containing preprocessing tokens A message is output if the arguments (before argument replacement) do not contain preprocessing tokens.
- (32) Macro replacement arguments with preprocessing directives A message is output if an argument list contains a preprocessor token stream that would function as a processing directive in another circumstance.
- (33) # operator

A message is output if the results of replacement are not a correct simple string literal.

(34) ## operator

A message is output if the results of replacement are not a correct simple string literal.

3.1.3 Processing System Dependent Items

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

(1) Environment

Table 3.1 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 3.2 Identifier Specifications

No.	Item	Compiler Specifications	
1	Number of valid letters in non externally-linked identifiers (inter- nal 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 3.3 Character Specifications

No.	Item	Compiler Specifications	
1	Elements of source character sets and execution envi- ronment character sets	Source program character sets and execu- tion 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 pro- gram execution	8 bits	
4	Relationship between source program character sets in character constants and strings and characters in exe- cution environment character sets	Corresponds to same ASCII characters.	
5	Values of integer character constants that include char- acters or escape sequences which are not stipulated in the language specifications	Characters and escape sequences which are not stipulated in the language specifica- tions 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 mes- sage 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 3.4Integer 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 respec- tively 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 nega- tive value	Maintains sign bit.

Table 3.5	Range of	Integer	Types and	Values

No.	Туре	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.

Notes 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 3.6Floating-Point Number Specifications

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

Table 3.7Limits of Floating-Point Type Values

		Limits	
No.	Item	Decimal Notation* ¹	Internal Representation (Hexadecimal)



п				
	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)	0000001
	3	Maximum values of double type and long double type ^{*2}	1.7976931348623158e+308 (1.7976931348623157e+308)	7fefffffffffff
	4	Minimum positive values of dou- ble type and long double type * ²	4.9406564584124655e-324 (4.9406564584124654e-324)	000000000000000000000000000000000000000

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.

Notes 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

ointer Specifications
ointer 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 3.9 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 3.10	Class, Structure,	Union, and Enumeration	ation Types, and	Bit Field Specifications
------------	-------------------	------------------------	------------------	--------------------------

No.	Item	Compiler Specifications
1	Referencing members in union type accessed by mem- bers 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), Com- pound 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 section3.1.4 Internal Data Representation and Areas.

Notes 2. Specifying the bit_order=left option assigns bit fields from the most significant bit.

Notes 3. When the **signed_bitfield** option is specified, the sign of bit fields is handled as the **signed int** type.

(9) Type Qualifiers

Table 3.11Type Qualifier Specifications

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

(10) Declarations

Table 3.12 Declaration Specifications

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

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

int a; Here, a has an int type (basic type) and the number of types modifying the basic type is 0.
 char *f(); Here, f has a function type returning a pointer type to a char type (basic type), and the num-

ber of types modifying the basic type is 2. (11) Statements

Table 3.13 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 3.14Preprocessor Specifications

No.	Item	Compiler Specifications
1	Relationship between single-character character con- stants 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.



Notes 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 of RX Build.

2	Method of reading include files	Files enclosed in "<" and ">" are read from the directory specified in the include option. If the file is not found, the search proceeds to the folder specified as the INC_RX envi- ronment variable.
3	Support for include files enclosed in double-quotes	Supported. The include file is read from the folder holding the file with the #include directive. If the file is not found, 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 3.2.3 #pragma Directive
6	DATEandTIMEvalues	Values are specified based on the host com- puter's timer at the start of compiling.

3.1.4 Internal Data Representation and Areas

This section explains the data type and the internal data representation. The internal data representation is determined according to the following four items:

- Size

Shows the memory size necessary to store the data.

- Boundary alignment

Restricts the addresses to which data is allocated. There are three types of alignment; 1-byte alignment in which data can be allocated to any address, 2-byte alignment in which data is allocated to even byte addresses, and 4-byte alignment in which data is allocated to addresses of multiples of four bytes.

- Data range

Shows the range of data of scalar type (C) or basic type (C++).

- Data allocation example

Shows an example of assignment of element data of compound type (C) or class type (C++).

(1) Scalar Type (C), Basic Type (C++)

Table 3.15 shows internal representation of scalar type data in C and basic type data in C++.

 Table 3.15
 Internal Representation of Scalar-Type and Basic-Type Data

N 0.	Data Type	Size (bytes	Align- ment	Sign	Data	Range
)	(bytes)		Minimum Value	Maximum Value
1	char* ¹	1	1	Unuse d	0	2 ⁸ –1 (255)
2	signed char	1	1	Used	-2 ⁷ (-128)	2 ⁷ –1 (127)
3	unsigned char	1	1	Unuse d	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	Unuse d	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	Unuse d	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)
12	unsigned long	4	4	Unuse d	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	Unus ed	0	2 ⁶⁴ –1 (18446744073709551615)
16	float	4	4	Used	-∞	+ ∞
17	double, long double	4* ⁴	4	Used	_ ∞	+ ∞
18	size_t	4	4	Unus ed	0	2 ³² –1 (4294967295)
19	ptrdiff_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	Unus ed	0	2 ³² –1 (4294967295)
22	bool* ⁵ _Bool* ⁸	1	1	*9		—
23	Reference* ⁶	4	4	Unus ed	0	2 ³² –1 (4294967295)
24	Pointer to a data mem- ber* 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.

Notes 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.

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



Notes 8. This data type is only valid for compilation in C99. The **_Bool** type is treated as the **bool** type in compilation.

Notes 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 3.16 shows internal representation of compound type and class type data.

Table 3. TO THICH AT TODIESCHILD TOT CONDULTUATION AND CLASS TYDE Data	Table 3.16	Internal	Representation of	Compound	Type and	Class Type Da	ata
--	------------	----------	-------------------	----------	----------	---------------	-----

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 Allo- cation, 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 Alloca- tion, below.	union { char a,b; }; Alignment: 1 byte Size: 1 byte
Class	 Always 4 if a virtual function is included Other than 1 above: maximum member align- ment 	Sum of data members, pointer to the virtual function table, and pointer to the virtual base class. Refer to (c) Class Data Alloca- tion, below.	<pre>class B:public A { virtual void f(); }; Alignment: 4 bytes Size: 8 bytes class A { char a; }; Alignment: 1 byte Size: 1 byte</pre>

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

st	crud	ct	: {
C	chai	2	a;
1	int	k);
}	ob	j	

	obj.a
obj.b	

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

obj.a	
	obj.b

(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;
```

0.a			
o.b[3]	o.b[2]	o.b[1]	o.b[0]
	o.b[6]	o.b[5]	o.b[4]

(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;
```

	obj.data1
obj.data2	

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;
```

obj.data3	obj.data2	obj.data1
-----------	-----------	-----------

For a class having a virtual base class, a pointer to the virtual base class is allocated.

Example

```
class A{
    short datal;
};
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 datal;
  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;
One byte
```



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





(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 3.17 shows the specifications of bit field members.

Table 3.17	Bit Field N	/lember	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 speci- fication	Unsigned. When the signed_bitfield option is specified, the signed type is selected.
4	Sign type for enum type	Signed. When the auto_enum option is specified, the resultant type is selected.

Notes 1. The **bool** type is only valid for compilation of C++ programs or C programs including **stdbool.h**.

Notes 2. To use a bit field member, data in the bit field is extended to the declared type. One-bit field data declared with a sign is interpreted as the sign, and can only indicate 0 and –1.

Notes 3. Zero extension: Zeros are written to the upper bits to extend data.

Notes 4. Sign extension: The most significant bit of a bit field is used as a sign and the sign is written to the upper bits to extend data.

Notes 5. This data type is only valid for programs in C99. The **_Bool** type is treated as the **bool** type in compilation.

(b) Bit Field Allocation

Bit field members are allocated according to the following five rules:

- Bit field members are placed in an area beginning from the right, that is, the least significant bit.

Example

struct b1 {
 int a:2;
 int b:3;
} x;



- Consecutive bit field members having type specifiers of the same size are placed in the same area as much as possible.

Example

```
struct bl {
   long a:2;
   unsigned int b:3;
} y;
```



- Bit field members having type specifiers with different sizes are allocated to separate areas.

Example

```
struct b1 {
    int a:5;
    char b:4;
} z;
```



- If the number of remaining bits in an area is less than the next bit field size, even though the type specifiers indicate the same size, the remaining area is not used and the next bit field is allocated to the next area.

Example

```
struct b2 {
   char a:5;
   char b:4;
} v;
```



- If a bit field member with a bit field size of 0 is declared, the next member is allocated to the next area.

Example

```
struct b2 {
    char a:5;
    char :0;
    char c:3;
} w;
```



- Note It is also possible to place bit field members from the upper bit. For details, refer to the description on the **bit_order** option in Compiler Options, and the description on **#pragma bit_order** in 3.2 Extended Language Specifications.
- (4) Memory Allocation in Big Endian

In big endian, data are allocated in the memory as follows:

- (a) One-Byte Data ((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 0x1	234 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 Address 0x101: 0x56 Address 0x102: 0x34 Address 0x103: 0x12 Big Endian: Address 0x100: 0x12 Address 0x101: 0x34 Address 0x102: 0x56 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

Address 0x101: 0x12 Address 0x102: Unused area Address 0x103: Unused area Address 0x104: 0xbc Address 0x105: 0x9a Address 0x106: 0x78 Address 0x107: 0x56 Big Endian: Address 0x100: 0x12 Address 0x101: 0x34 Address 0x102: Unused area Address 0x103: Unused area Address 0x104: 0x56 Address 0x105: 0x78 Address 0x106: 0x9a 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 Address 0x101: 0x00 Address 0x102: 0x01 Address 0x103: 0x00 Address 0x104: 0x01 Address 0x105: 0x00 Address 0x106: Unused area Address 0x107: Unused area

Big Endian: Address 0x100: 0x00 Address 0x101: 0x01 Address 0x102: 0x00 Address 0x103: 0x01 Address 0x104: 0x00 Address 0x105: 0x01 Address 0x106: Unused area Address 0x107: Unused area

- (5) Floating-Point Number Specifications
 - (a) Internal Representation of Floating-Point Numbers Floating-point numbers handled by this compiler are internally represented in the standard IEEE format. This section outlines the internal representation of floating-point numbers in the IEEE format. This section assumes that the dbl_size=8 option is specified. When the dbl_size=4 option is specified, the internal representation of the double type and long double type is the same as that of the float type.
 - (b) Format for Internal Representation

float types are represented in the IEEE single-precision (32-bit) format, while double types and long double types are represented in the IEEE double-precision (64-bit) format.

Structure of Internal Representation (C)

Figure 3.1 shows the structure of the internal representation of **float**, **double**, and **long double** types.





double type and long double type



The internal representation format consists of the following parts:

- i. Sian
 - 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.

Types of Values Represented as Floating-Point Numbers (d)

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.

- Normalized number i.
- 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 3.18 shows the types of values represented as floating-point numbers.

Mantissa	Exponent		
	0	Not 0 or Not All Bits are 1	All Bits are 1
0	0	Normalized number	Infinity
Other than 0	Denormalized number		Not-a-number

Note

Denormalized numbers are floating-point numbers of small absolute values that are outside the range represented by normalized numbers. There are fewer valid digits in a denormalized number than in a normalized number. Therefore, if the result or intermediate result of a calculation is a denormalized number, the number of valid digits in the result cannot be guaranteed. When denormalize=off is specified, denormalized numbers are processed as 0. When denormalize=on is specified, denormalized numbers are processed as denormalized numbers

- float Type (e)
 - The float type is internally represented by a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa.
 - Normalized numbers i.

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: $(-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times (1 + (\text{mantissa}) \times 2^{-23})$

Example

30 31 23 22 1 10000000 Sign: Exponent: $1000000_{(2)} - 127 = 1$, where $_{(2)}$ indicates binary Mantissa: $1.11_{(2)} = 1.75$ $-1.75 \times 2^{1} = -3.5$ Value: 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 2²³rd bit is 0 and this bit is followed by the decimal point. Values of denormalized numbers are as follows:

 $(-1)^{\text{sign}} \times 2^{-126} \times ((\text{mantissa}) \times 2^{-23})$

Example

31	30	23 22				
C	0000000	0 110	000000000000000000000000000000000000000	00000000		
	Sign: Exponent: Mantissa:	+ -126 0.11 ₍₂) = 0.75, where (₂₎ indicates l	inary	
	Value: Zero	0.75 >	< 2 ⁻¹²⁶			
iv.	The sign is both 0. +0.0 and –0 Infinity	0 (positi).0 are b	ve) or 1 (negativ oth the value 0.0	e), indicating).	+0.0 or –0.0, respectively	. The exponent and mantissa are
v.	The sign is The expone The mantiss Not-a-nur The expone The mantiss	0 (positi ent is 25 sa is 0. mber ent is 25 sa is a v	ve) or 1 (negativ 5 (2 ⁸ –1). 5 (2 ⁸ –1). alue other than (e), indicating	+∞ or –∞, respectively.	
No	ote	A not-a when th the ma	n-number is calle he MSB of the m ntissa and of the	d a quiet Na antissa is 0. sign.	N when the MSB of the ma There are no stipulations r	intissa is 1, or a signaling NaN regarding the values of the rest of
do Th ma i.	uble Types e double a antissa. Normalize	and long nd long ed numt	g double Types double types a pers	re internally	epresented by a 1-bit sign	, an 11-bit exponent, and a 52-bit
	The sign inc 2046 (2 ¹¹ –2 and 1023. T 2 ⁵² nd bit is (–1) ^{sign} × 2 [€]	licates t !). The a he man 1 and th exponent-	he sign of the va ctual exponent is tissa is between his bit is followed $^{1023} \times (1+(manti$	lue, either 0 s gained by s 0 and 2^{52} by the decin ssa) × 2 ⁻⁵²)	(positive) or 1 (negative). ubtracting 1023 from this v . The actual mantissa is in nal point. Values of normal	The exponent is between 1 and alue. The range is between –1022 terpreted as the value of which ized numbers are as follows:

Example

(f)

63 62 52 51 Sian: Exponent: $111111111_{(2)} - 1023 = 0$, where (2) indicates binary Mantissa: 1.111₍₂₎ = 1.875 $1.875 \times 2^{0} = 1.875$ Value:

RENESAS

```
ii. Denormalized numbers
```

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is 0 and the actual exponent is -1022. The mantissa is between 1 and $2^{52}-1$, and the actual mantissa is interpreted as the value of which 2^{52} nd bit is 0 and this bit is followed by the decimal point. Values of denormalized numbers are as follows:

 $(-1)^{sign} \times 2^{-1022} \times ((mantissa) \times 2^{-52})$

Example

63 62 52 51 Sign: Exponent: -1022 Mantissa: 0.111₍₂₎ = 0.875, where $_{(2)}$ indicates binary Value: 0.875 \times 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. Infinity iv. The sign is 0 (positive) or 1 (negative), indicating $+\infty$ or $-\infty$, respectively. The exponent is 2047 (2^{11} -1). The mantissa is 0. Not-a-number V. The exponent is 2047 (2¹¹–1). The mantissa is a value other than 0. A not-a-number is called a quiet NaN when the MSB of the mantissa is 1, or signaling NaN when Note the MSB of the mantissa is 0. There are no specifications regarding the values of other mantissa

3.1.5 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 3.19 shows each operator precedence and associativity.

fields or the sign.

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	1	Left	Bitwise inclusive OR expression
12	&&	Left	Logical AND operation

Table 3.19 Operator Precedence and Associativity



13	II	Left	Logical inclusive OR expression
14	?:	Right	Conditional expression
15	= += -= *= /= %= <<= >>= &= = ^=	Right	Assignment expression
16	3	Left	Comma expression

3.1.6 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

3.2 Extended Language Specifications

This section explains the extended language specifications supported by the CCRX. The compiler supports the following extended specifications:

- **#pragma** extension specifiers and keywords
- Intrinsic functions
- Section address operators

3.2.1 Macro Names

The following shows supported macro names.



Table 3.20 Predefined Macros of Compiler

No.	Option	Predefined Macro		
		DATE	Date of translating source file (character string constant in the form of "Mmm dd yyyy".) Here, the name of the month is the same as that created by the asctime func- tion stipulated by ANSI standards (3 alphabetic characters with only the first character is capital letter) (The first character of dd is blank if its value is less than 10).	
	_	_FILE_	Name of assumed source file (character string constant).	
	_	_LINE_	Line number of source line at that point (decimal).	
	_	_STDC_	1	
	lang=c99	_STDC_HOSTED_	1	
	lang=c* ¹ lang=c99	_STDC_VERSION_	199409L (lang = c* ¹) 199901L (lang = c99)	
	lang=cpp* ² lang=ecpp	_cplusplus	1	
		TIME	Translation time of source file (character string constant having format "hh:mm:ss").	
1	isa=rxv1 isa=rxv2	#defineRXV1 #defineRXV2	1 1	
2	endian=big endian=little	#defineBIG #defineLIT	1 1	
3	dbl_size=4 dbl_size=8	#defineDBL4 #defineDBL8	1 1	
4	int_to_short	#define INT_SHORT	1	
5	signed_char unsigned_char	#defineSCHAR #defineUCHAR	1 1	
6	signed_bitfield unsigned_bitfield	#defineSBIT #defineUBIT	1 1	
7	round=zero round=nearest	#defineROZ #defineRON	1 1	
8	denormalize=off denormalize=on	#defineDOFF #defineDON	1 1	
9	bit_order=left bit_order=right	#defineBITLEFT #defineBITRIGHT	1 1	
10	auto_enum	#defineAUTO_ENUM	1	
11	library=function library=intrinsic	#defineFUNCTION_LIB #defineINTRINSIC_LIB	1	
12	fpu	#defineFPU	1	
13	—	#defineRENESAS* ³	1	



No.	Option	Predefined Macro	
14	—	#defineRENESAS_VERSION* ³	0xAABBCC00* ⁴
15	_	#defineRX* ³	1
16	pic	#definePIC	1
17	pid	#definePID	1
18	cpu=rx600 cpu=rx200	#defineRX600 #defineRX200	1 1

Notes 1. Includes cases where a file with the **.c** extension is compiled without specifying the **-lang** option.

- Notes 2. Includes cases where a file with the **.cpp**, **.cp**, or **.cc** extension is compiled without specifying the **-lang** option.
- Notes 3. Always defined regardless of the option.
- Notes 4. When the version is V.AA.BB.CC, the value of **___RENESAS_VERSION__** is 0xAABBCC00. Example: For V.2.02.00, specify **#define ___RENESAS_VERSION__0x02020000**.

3.2.2 Keywords

The CCRX adds the following characters as a keyword to implement the extended function. These words are similar to the ANSI C keywords, and cannot be used as a label or variable name.

Keywords that are added by the CCRX are listed below.

__evenaccess, far, _far, near, and _near

Table 3.21 Keywords

No.	Keyword	Function
1	#pragma STDC CX_LIMITED_RANGE #pragma STDC FENV_ACCESS #pragma STDC FP_CONTRACT	Reserved keywords that are only valid when C99 is selected (these are only for grammatical checking and not for checking the correctness of the code).
2	#pragma keywords	Provides language extensions. For details, refer to 3.2.3 #pragma Directive.
3	evenaccess	Guarantees access in the size of the variable type.
4	far _far near _near	Reserved keywords (these are ignored even though they are rec- ognized as type names)
5	_RAM_BASE	Reserved keyword. (Only in -base=ram sepcified.)
6	_ROM_BASE	Reserved keyword. (Only in -base=rom sepcified.)
7	_PID_TOP	Reserved keyword. (Only in -pid sepcified.)
8	_builtin_xxx	Reserved keyword. This means all functions those names begin with _builtin

3.2.3 #pragma Directive

(1) Section Switch

This extension changes the section name to be output by the compiler. For details on the description, refer to(1) Section Switch.

<pre>#pragma section [<section type="">] [D<new name="" section="">]</new></section></pre>
<section type="">: { P C D B }</section>

 (2) Stack Section Creation This extension creates the stack section.
 For details on the description, refer to (2) Stack Section Creation.

#pragma stacksize { si=<constant> | su=<constant> }

Interrupt Function Creation
 This extension creates the interrupt function.
 For details on the description, refer to (3) Interrupt Function Creation.

#pragma interrupt [(]<function name>[(<interrupt specification>[,...])][,...][)]

 (4) Inline Expansion of Function This extension expands a function. For details on the description, refer to (4) Inline Expansion of Function.

#pragma inline [(]<function name>[,...][)]

 (5) Cancellation of Inline Expansion of Function This extension cancels expansion of a function. For details on the description, refer to (4) Inline Expansion of Function.

#pragma noinline [(]<function name>[,...][)]

 Inline Expansion of Assembly-Language Function This extension creates the assembly-language inline functions.
 For details on the description, refer to (5) Inline Expansion of Assembly-Language Function.

#pragma inline_asm[(]<function name>[,...][)]

 (7) Entry Function Specification This extension specifies the entry function. For details on the description, refer to (6) Entry Function Specification.

#pragma entry[(]<function name>[)]

Bit Field Order Specification
 This extension specifies the order of the bit field.
 For details on the description, refer to (7) Bit Field Order Specification.

#pragma bit_order [{left | right}]

(9) 1-Byte Alignment Specification for Structure Members and Class Members This extension specifies the boundary alignment value of structure members and class members as 1 byte. For details on the description, refer to (8) Alignment Value Specification for Structure Members and Class Members.

#pragma pack

- (10) Default Alignment Specification for Structure Members and Class Members
 - This extension specifies the boundary alignment value for structure members and class members as the value for members.

For details on the description, refer to (8) Alignment Value Specification for Structure Members and Class Members.

#pragma unpack

(11) Option Alignment Specification for Structure Members and Class Members
 This extension specifies the option of the boundary alignment value for structure members and class members.

For details on the description, refer to (8) Alignment Value Specification for Structure Members and Class Members.

#pragma packoption

(12) Allocation of a Variable to the Absolute AddressThis extension allocates the specified variable to the specified address.For details on the description, refer to (9) Allocation of a Variable to the Absolute Address.

#pragma address [(]<variable name>=<absolute address>[,...][)]

(13) Endian Specification for Initial Values
 This extension specifies an endian for initial values.
 For details on the description, refer to (10) Endian Specification for Initial Values.

#pragma endian [{big | little}]

(14) Specification of Function in which Instructions at Branch Destinations are Aligned to 4-Byte Boundaries. This extension specifies the function in which instructions at branch destinations are aligned to 4-byte boundaries. For details on the description, refer to (11) Specification of Function in which Instructions at Branch Destinations are Aligned for Execution.

#pragma instalign4 [(]<function name>[(<branch destination type>)][,...][)]

(15) Specification of Function in which Instructions at Branch Destinations are Aligned to 8-Byte Boundaries. This extension specifies the function in which instructions at branch destinations are aligned to 8-byte boundaries. For details on the description, refer to (11) Specification of Function in which Instructions at Branch Destinations are Aligned for Execution.

#pragma instalign8 [(]<function name>[(<branch destination type>)][,...][)]

 (16) Specification of Function in which Instructions at Branch Destinations are not Aligned This extension specifies the function in which instructions at branch destinations are not aligned. For details on the description, refer to (11) Specification of Function in which Instructions at Branch Destinations are Aligned for Execution.

#pragma noinstalign [(]<function name>[,...][)]

3.2.4 Using Extended Specifications

This section explains using the following extended specifications.

- Section switch
- Stack section creation
- Interrupt function creation
- Inline expansion of function
- Inline expansion of assembly-language function
- Entry function specification
- Bit field order specification
- Alignment value specification for structure members and class members
- Allocation of a variable to the absolute address
- Endian specification for initial values
- Specification of function in which instructions at branch destinations are aligned for execution
- (1) Section Switch

#pragma section [<section type>] [D<new section name>]
<section type>: { P | C | D | B }



This extension changes the section name to be output by the compiler.

When both a section type and a new section name are specified, the section names for all functions written after the **#pragma** declaration are changed if the specified section type is **P**. If the section type is **C**, **D**, or **B**, the names of all sections defined after the **#pragma** declaration are changed.

When only a new section name is specified, the section names for the program, constant, initialized data, and uninitialized data areas after the **#pragma** declaration are changed. In this case, the default section name post-fixed with the string specified by <new section name> is used as the new section name.

When neither a section type nor a new section name is specified, the section names for the program, constant, initialized data, and uninitialized data areas after the **#pragma** declaration are restored to the default section names. The default section name for each section type is determined by the **section** option when specified. If the default section name is not specified by the **section** option, the section type name is used instead.

Examples 1. When a section name and a section type are specified

#pragma section B Ba
int i; // Allocated to the Ba section
void func(void)
{

(omitted)

}

#pragma section B Bb
int j; // Allocated to the Bb section
void sub(void)
{
 (omitted)

Examples 2. When the section type is omitted

<pre>#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 {</pre>	
<pre>#pragma section int b;// Allocated to the B section void g(void)// Allocated to the P section { b=c; }</pre>	

#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 and initializers for use in the dynamic initialization of aggregates

(2) Branch table of **switch** statement

Up to 2045 sections can be specified by **#pragma section** in one file.

When specifying the section for static class member variables, be sure to specify **#pragma section** for both the class member declaration and definition.



Example

/*	
** Clas */	ss member declaration
, 	
CI2 {	ISS A
	private:
	// No initial value specified
;	#pragma section DATA
static ir #pragm	nt data_; na section
// Initial #pragm	value specified na section TABLE
static ir	nt table_[2];
#pragn	na section
ر	
/* **	Class member definition
*/	
// •	
// 1 #p	ragma section DATA
int	A::data_;
#р	ragma section
// 1	nitial value specified
#p int	ragma section TABLE A::table [2]={0, 1}:
#p	ragma section

(2) Stack Section Creation

#pragma stacksize { si=<constant> | su=<constant> }

When si=<constant> is specified, a data section is created to be used as the stack of size <constant> with section name SI.

When su=<constant> is specified, a data section is created to be used as the stack of size <constant> with section name SU.

C source description:

#pragma stacksize si=100 #pragma stacksize su=200

Example of expanded code:

.SECTION SI,DATA,ALIGN=4 .BLKB 100 .SECTION SU,DATA,ALIGN=4 .BLKB 200

si and su can each be specified only once in a file. <constant> must always be specified as a multiple of four. A value from 4 to 2147483644(0x7fffffc) is specifiable for <constant>.

(3) Interrupt Function Creation

#pragma interrupt [(]<function name>[(<interrupt specification>[,...])][,...][)]

This extension declares an interrupt function.

A global function or a static function member can be specified for the function name. Table 3.22 lists the interrupt specifications.

Table 3.22	Interrupt Specifications
------------	--------------------------

No.	Item	Form	Options	Specifications
1	Vector table	vect=	<vector number=""></vector>	Specifies the vector number for which the interrupt function address is stored.
2	Fast interrupt	fint	None	Specifies the function used for fast inter- rupts. 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 begin- ning of the function to enable nested interrupts.
5	Accumulator saving	acc	None	Saves and restores Accumulator in the interrupt function.
6	Accumulator non-sav- ing	no_acc	None	Does not save and restore Accumulator 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 Accumulator 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. The save and restored code of the **ACC** when the ISA ^{*1} is selected as the RXv1 or the microcomputer type is selected by the CPU ^{*2}.

When Accumulator non-saving is specified (no_acc), an instruction to save and restore the ACC is not generated.

If neither **acc** nor **no_acc** is specified, the result depends on the 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.

Note

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

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

Examples 1. Correct declaration and wrong declaration

#pragma interrupt (f1, f2)
void f1(){...}// Correct declaration.
int f2(){...}// An error will be output
 // because the return value is not
 // void data.

Examples 2. General interrupt function

C source description: #pragma interrupt func void func(){ }

Output code:

_func: PUSHM R1-R3; Saves the registers used in the function. (R1, R2, and R3 are used in the function) POPM R1-R3; Restores the registers saved at the entry. RTE

Examples 3. Interrupt function that calls another function

In addition to the registers used in the interrupt function, the registers that are not guaranteed before and after a function call are also saved at the entry and restored at the exit.

C source description:

#pragma interrupt func
void func(){
 ...
 sub();
 ...
}

Output code:



Examples 4. Use of interrupt specification fint C source description: Compiles with the fint_register=2 option specified

#pragma interrupt func1(fint)
void func1(){ a=1; } // Interrupt function
void func2(){ a=2; } // General function

Output code:



_func1: PUSHM R1-R3; Saves the registers used in the function. ;(Note that R12 and R13 are not saved.)		
 (R1, R2, R3, R12, and R13 are used in the function.)		
 POPM R1-R3 ; Restores the registers saved at the entry. RTFI		
_func2: ; In the functions without #pragma interrupt fint ; specification, do not use R12 and R13. RTE		

Examples 5. Use of interrupt specification acc C source description:

void func5(void);
#pragma interrupt accsaved_ih(acc) /* Specifies acc */
void accsaved_ih(void)
{

func5();

Output code:

_accsaved_ih: PUSHM R14-R15 PUSHM R1-R5 MVFACMI R4 SHLL #10H, R4 MVFACHI R5 PUSHM R4-R5 BSR_func5 POPM R4-R5 MVTACLO R4 MVTACHI R5 POPM R1-R5 POPM R14-R15		
POPM R14-R15 RTF		
POPM R14-R15 RTE		

[Remarks]

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.

Parameters are not definable for **#pragma interrupt** functions. Although defining parameters for such functions does not lead to an error, values read out from the parameters are undefined.

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.

(4) Inline Expansion of Function

#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;
```

#pragma inline [(]<function name>[,...][)
#pragma noinline [(]<function name>[,...][)]

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.

The C++ compiler does not create external definitions for inline-specified functions.

The C (C99) does not create external definitions for inline-specified functions unless they include **extern** declarations.

(5) Inline Expansion of Assembly-Language Function

#pragma inline_asm[(]<function name>[,...][)]

This extension declares an assembly-language function for which inline expansion is performed. The general function calling rules are also applied to the calls of assembly-language inline functions.

Example C source description:


#pragma inline_asm func
static int func(int a, int b){
 ADD R2,R1; Assembly-language description
}
main(int *p){
*p = func(10,20);
}

Output code:

_main: PUSH.L R6 MOV.L R1, R6 MOV.L #20, R2 MOV.L #10, R1 ADD R2,R1; Assembly-language description MOV.L R1, [R6] MOV.L #0, R1 RTSD #04H, R6-R6

Specify #pragma inline_asm before defining a function.

An external definition is generated 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 3.26 Rules to Use Registers) are used in an assembly-language inline function, these registers must be saved and restored at the start and end of the function.

[Remarks]

- In an assembly-language inline function, use only the RX Family instruction and temporary labels. Other labels cannot be defined and assembler directives cannot be used.
- Do not use **RTS** at the end of an assembly-language inline function.
- Function members cannot be specified as function names.
- When **#pragma inline_asm** is specified for a **static** function, the function definition is deleted after inline expansion.
- Assembly-language descriptions are processed by the preprocessor; take special care when defining through **#define** a macro with the same name as an instruction or a register used in the assembly language (such as **MOV** or **R5**).
- A stack information file handles the assembly code for a **#pragma inline_asm** directive as not consuming stack area. Be careful when the assembly code includes an instruction with **R0** as an operand.

-

(6) Entry Function Specification

#pragma entry[(]<function name>[)]

This specifies that the function specified as <function name> is handled as an entry function.

The entry function is created without any code to save and restore the contents of registers.

When **#pragma stacksize** is declared, the code that makes the initial setting of the stack pointer will be output at the beginning of the function.

When the **base** option is specified, the base register specified by the option is set up.

Example C source description: -base=rom=R13 is specified

#pragma stacksize su=100 #pragma entry INIT
void INIT() {
:
•

Output code:



.SECTION SU,DATA,ALIGN=4 .BLKB 100 .SECTION P,CODE _INIT: MVTC (TOPOF SU + SIZEOF SU),USP MOV.L #__ROM_TOP,R13

Be sure to specify **#pragma entry** before declaring a function. Do not specify more than one entry function in a load module.

(7) Bit Field Order Specification

#pragma bit_order [{left | right}]

This extension switches the order of bit field assignment.

When **left** is specified, bit field members are assigned from the upper-bit side. When **right** is specified, members are assigned from the lower-bit side.

The default is right.

If left or right is omitted, the order is determined by the option specification.

Example



(8) Alignment Value Specification for Structure Members and Class Members

toragma pack	
toragma unpack	
tpragma packoption	

#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 3.24 shows **#pragma pack** specifications and the corresponding alignment values.

Table 3.23	#pragma pack	Specifications an	nd Corresponding	Member Alignment Values
				J

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

<pre>#pragma pack struct S1 { char a;/* Byte offset = 0*/ int b;/* Byte offset = 1*/ char c;/* Byte offset = 5*/</pre>	
} ST1;/* Total size: 6 bytes*/	
<pre>#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*/</pre>	

The boundary alignment value for structure and class members can also be specified by the **pack** option. When both the option and **#pragma** extension specifier are specified together, the **#pragma** specification takes priority.

(9) Allocation of a Variable to the Absolute Address

#pragma address [(]<variable name>=<absolute address>[,...][)]

This extension allocates the specified variable to the specified address. The compiler assigns a section for each specified variable, and the variable is allocated to the specified absolute address during linkage. If variables are specified for contiguous addresses, these variables are assigned to a single section.

Example C source description:

#pragma address X=0x7f00
int X;
main(){
 X=0;
}

Output code:



```
main:
   MOV.L
            #0.R5
   MOV.L
             #7F00H,R14;
   MOV.L
             R5,[R14]
   RTS
   .SECTION $ADDR B 7F00
   .ORG
            7F00H
   .glb
           Х
     ; static: X
X:
   .blkl
          1
```

[Remarks]

-Specify **#pragma address** before declaring a variable.

- If an object that is neither a structure/union member nor a variable is specified, an error will be output.

- If **#pragma address** is specified for a single variable more than one time, an error will be output.

- A static variable that is validated by #pragma address and not referred from the source file may be removed by a compiler optimization.

- We recommend not applying #pragma address to a variable which has an initial value but does not have the const qualifier. If this case applies for any variables, take note of the restrictions below.

- The -rom option (RAMization of the ROM area) of the optimizing linkage editor (rlink) cannot be applied to sections containing such variables.

- Error messages or warnings will not be displayed for code that includes such a variable.

- When a section containing such variables is allocated to the RAM, all initial values must be written to the corresponding RAM areas when starting up the program or in advance of that.

(10) Endian Specification for Initial Values

#pragma endian [{big | little}]

This extension specifies the endian for the area that stores static objects.

The specification of this extension is applied from the line containing **#pragma endian** to the end of the file or up to the line immediately before the line containing the next **#pragma endian**.

big specifies big endian. When the **endian=little** option is specified, data is assigned to the section with the section name postfixed with **_B**.

little specifies little endian. When the **endian=big** option is specified, data is assigned to the section with the section name postfixed with _L.

When **big** or **little** is omitted, endian is determined by the option specification.

Example When the **endian=little** option is specified (default state)

C source description:

#pragma endian big
int A=100;/* D_B section */
#pragma endian
int B=200;/* D section */

Output code:

.glb _A .glb _B .SECTION D,ROMDATA,ALIGN=4 _B: .lword 200 .SECTION D_B,ROMDATA,ALIGN=4 .ENDIAN BIG _A: .lword 100

If areas of the **long long** type, **double** type (when the **dbl_size=8** option is specified), and **long double** type (when the **dbl_size=8** option is specified) are included in objects to which **#pragma endian** (differed from the **endian** option) is applied, do not make indirect accesses to these areas using addresses or pointers. In such a



case, correct operation will not be guaranteed. If a code that acquires an address in such an area is included, a warning message is displayed.

If bit fields of the **long long** type are included in objects to which **#pragma endian** (differed from the **endian** option) is applied, do not make writes to these areas. In such a case, correct operation will not be guaranteed. 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 and initializers for use in the dynamic initialization of aggregates

(2) Branch table of switch statement

(3) Objects declared as external references (objects declared through **extern** without initialization expression)

(4) Objects specified as **#pragma address**

(11) Specification of Function in which Instructions at Branch Destinations are Aligned for Execution

#pragma instalign4 [(]<function name>[(<branch destination type>)][,...][)]
#pragma instalign8 [(]<function name>[(<branch destination type>)][,...][)]
#pragma noinstalign [(]<function name>[,...][)]

Specifies the function in which instructions at branch destinations are aligned for execution. Instruction allocation addresses in the specified function are adjusted to be aligned to 4-byte boundaries when **#pragma instalign4** is specified or to 8-byte boundaries when **#pragma instalign8** is specified. In the function specified with **#pragma noinstalign**, alignment of allocation addresses is not adjusted. The branch destination type should be selected from the following*: No specification:Head of function and **case** and **default** labels of **switch** statement inmostloop: Head of each inmost loop, head of function, and **case** and **default** labels of **switch** statement

loop: Head of each loop, head of function, and case and default labels of switch statement

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

Except that each **#pragma** extension specification is valid only in the specified function, these specifiers work in the same way as the **instalign4**, **instalign8**, and **noinstalign** options. When both the options and **#pragma** extension specifiers are specified together, the **#pragma** specifications take priority.

In the code section that contains a function specified with **instalign4** or **instalign8**, the alignment value is changed to 4 (**instalign4** is specified) or 8 (**instalign8** is specified). If a single code section contains both a function specified with **instalign4** and that specified with **instalign8**, the alignment value in the code section is set to 8. The other detailed functions of these **#pragma** extension specifiers are the same as those of the **instalign4**, **instalign8**, and **noinstalign** options; refer to the description of each option.

3.2.5 Using a Keyword

This section explains using the following keyword.

- Description of access in specified size
- (1) Description of Access in Specified Size

__evenaccess <type specifier> <variable name> <type specifier> __evenaccess <variable name>

A variable is accessed in the declared or defined size.

This extension guarantees access in the size of the target variable.

Access size is guaranteed for 4-byte or smaller scalar integer types (signed char, unsigned char, signed short, unsigned int, unsigned int, signed long, and unsigned long).

Example C source description:



Output code (__evenaccess not specified):

_test: MOV.L #16712056,R1 BCLR #5,[R1] ; Memory access in 1 byte RTS

Output code (__evenaccess specified):

_test: MOV.L #16712056,R1 MOV.L [R1],R5 ; Memory access in 4 bytes BCLR #5,R5 MOV.L R5,[R1] ; Memory access in 4 bytes RTS

The __evenaccess is invalid to the case of accessing of members by a lump of these structure and union frame. When __evenaccess is specified for a structure or a union, __evenaccess is applied to all members. In this case, the access size is guaranteed for 4-byte or smaller scalar integer types, but the size of access in structure or union units is not guaranteed.

3.2.6 Intrinsic Functions

In the CCRX, some of the assembler instructions can be described in C source as "Embedded Functions". However, it is not described "as assembler instruction", but as a function format set in the CCRX. When these functions are used, output code outputs the compatible assembler instructions without calling the ordinary function.

No.	Item	Specifications	Function	Restriction in User Mode*
1	Maximum value and minimum	signed long max(signed long data1, signed long data2)	Selects the maximum value.	0
2	value	signed long min(signed long data1, signed long data2)	Selects the minimum value.	0
3	Byte switch	unsigned long revl(unsigned long data)	Reverses the byte order in longword data.	0
4		unsigned long revw(unsigned long data)	Reverses the byte order in longword data in word units.	0
5	Data exchange	void xchg(signed long *data1, signed long *data2)	Exchanges data.	0

Table 3.24 Intrinsic Functions



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 *addr2)	Multiply-and-accumulate operation (byte).	0
7		long long rmpaw(long long init, unsigned long count, short *addr1, short *addr2)	Multiply-and-accumulate operation (word).	0
8		long long rmpal(long long init, unsigned long count, long *addr1, long *addr2)	Multiply-and-accumulate operation (longword).	0
9	Rotation	unsigned long rolc(unsigned long data)	Rotates data including the carry to left by one bit.	0
10		unsigned long rorc(unsigned long data)	Rotates data including the carry to right by one bit.	0
11		unsigned long rotl(unsigned long data, unsigned long num)	Rotates data to left.	0
12		unsigned long rotr(unsigned long data, unsigned long num)	Rotates data to right.	0
13	Special instruc- tions	void brk(void)	BRK instruction excep- tion.	0
14		void int_exception(signed long num)	INT instruction exception.	0
15		void wait(void)	Stops program execution.	×
16		void nop(void)	Expanded to a NOP instruction.	0
17	Processor inter- rupt priority	void set_ipl(signed long level)	Sets the interrupt priority level.	×
18	level (IPL)	unsigned char get_ipl(void)	Refers to the interrupt pri- ority level.	0
19	Processor sta-	void set_psw(unsigned long data)	Sets a value for PSW .	Δ
20	lus word (PSVV)	unsigned long get_psw(void)	Refers to PSW value.	0
21	Floating-point	void set_fpsw(unsigned long data)	Sets a value for FPSW .	0
22	(FPSW)	unsigned long get_fpsw(void)	Refers to FPSW value.	0
23	User stack	void set_usp(void * data)	Sets a value for USP .	0
24	pointer (USP)	void * get_usp(void)	Refers to USP value.	0
25	Interrupt stack	void set_isp(void * data)	Sets a value for ISP .	Δ
26	pointer (ISP)	void * get_isp(void)	Refers to ISP value.	0
27	Interrupt table	void set_intb(void * data)	Sets a value for INTB .	Δ
28	ายนายเยเลา (แก่ เ ย)	void * get_intb(void)	Refers to INTB value.	0
29	Backup PSW	void set_bpsw(unsigned long data)	Sets a value for BPSW .	Δ
30		unsigned long get_bpsw(void)	Refers to BPSW value.	0
31	Backup PC	void set_bpc(void * data)	Sets a value for BPC .	Δ
32		void * get_bpc(void)	Refers to BPC value.	0



No.	Item	Specifications	Function	Restriction in User Mode*
33	Fast interrupt	void set_fintv(void * data)	Sets a value for FINTV .	Δ
34	vector register (FINTV)	void * get_fintv(void)	Refers to FINTV value.	0
35	Significant 64-bit multipli-	signed long long emul(signed long data1, signed long data2)	Signed multiplication of significant 64 bits.	0
36	cation	unsigned long long emulu(unsigned long data1, unsigned long data2)	Unsigned multiplication of significant 64 bits.	0
37	Processor mode (PM)	void chg_pmusr(void)	Switches to user mode.	Δ
38	Accumulator	void set_acc(signed long long data)	Sets the ACC.	0
39	(ACC)	signed long long get_acc(void)	Refers to the ACC.	0
40	Control of the interrupt enable	void setpsw_i(void)	Sets the interrupt enable bit to 1.	Δ
41 Dits		void clrpsw_i(void)	Clears the interrupt enable bit to 0.	Δ
42	Multiply-and- accumulate	long <u>macl</u> (short* data1, short* data2, unsigned long count)	Multiply-and-accumulate operation of 2-byte data.	0
43	operation	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.	0
44	Exception vec-	void set_extb(void * data)	Sets a value for EXTB .	Δ
45	tor table register (EXTB)	void * get_extb(void)	Refers to EXTB value.	0

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.

 $\Delta\!\!:$ Has no effect when executed in user mode.

signed long max(signed long data1, signed long data2)

[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)



```
[Description]
Selects the smaller of two input values (this function is expanded into a MIN instruction).
[Header]
<machine.h>
[Parameters]
         Input value 1
data1
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)

```
[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)

```
[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)

[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 *addr2)

[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]
initInitial value
countCount of multiply-and-accumulate operations
*addr1Start address of values 1 to be multiplied
*addr2Start address of values 2 to be multiplied
[Return value]
Lower 64 bits of the init + \Sigma(data1[n] * data2[n]) result (n = 0, 1, ..., const - 1)
[Example]
 #include <machine.h>
 extern signed char data1[8],data2[8];
 long long sum;
 void main(void)
 {
      sum=rmpab(0, 8, data1, data2);
        // Specifies 0 as the initial value, adds the result
        // of multiplication of arrays data1 and data2,
        // and stores the result in sum.
 }
```

[Remark]

The RMPA instruction obtains a result in a maximum of 80 bits, but this intrinsic function handles only 64 bits.

```
long long rmpaw(long long init, unsigned long count, short *addr1, short *addr2)
```

[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 + <math>\Sigma(data1[n] * data2[n]) result (n = 0, 1, ..., const - 1) 
[Example]
```



```
#include <machine.h>
extern signed short data1[8],data2[8];
long long sum;
void main(void)
{
    sum=rmpaw(0, 8, data1, data2);
    // Specifies 0 as the initial value, adds the result
    // of multiplication of arrays data1 and data2,
    // and stores the result in sum.
}
```

[Remark]

The **RMPA** instruction obtains a result in a maximum of 80 bits, but this intrinsic function handles only 64 bits.

long long rmpal(long long init, unsigned long count, long *addr1, long *addr2)

[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]
        Initial value
init
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 + \Sigma(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)

```
[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)

[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)

[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]



unsigned long rotr(unsigned long data, unsigned long num)

[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)

[Description] This function is expanded into a BRK instruction. [Header] <machine.h> [Parameters] -[Return value] -

[Example]

```
#include <machine.h>
void main(void)
{
    brk();// BRK instruction
}
```

void int_exception(signed long num)

[Description] This function is expanded into an INT num instruction. [Header] <machine.h> [Parameters] num INT instruction number [Return value]

[Example]



```
#include <machine.h>
```

```
void main(void)
```

{

```
int_exception(10);// INT #10 instruction
```

}

[Remarks]

Only an integer from 0 to 255 can be specified as num.

void wait(void)

[Description] This function is expanded into a WAIT instruction. [Header] <machine.h> [Parameters]

[Return value]

[Example]

```
#include <machine.h>
void main(void)
{
    wait();// WAIT instruction
```

}

[Remarks]

This function must not be executed when the RX processor mode is user mode. If executed, a privileged instruction exception of the RX occurs due to the specifications of the **WAIT** instruction.

void nop(void)

```
[Description]

This function is expanded into a NOP instruction.

[Header]

<machine.h>

[Parameters]

-

[Return value]

-

[Example]

#include <machine.h>
```

```
void main(void)
{
     nop();// NOP instruction
}
```

```
void set_ipl(signed long level)
```

[Description] Changes the interrupt mask level. [Header] <machine.h> [Parameters]

[Return value] level Interrupt mask level to be set [Example]



```
#include <machine.h>
void main(void)
{
    set_ipl(7);// Sets PSW.IPL to 7.
}
```

[Remarks]

A value from 0 to 15 can be specified for level by default, and a value from 0 to 7 can be specified when -patch=rx610 is specified.

If a value outside the above range is specified when level is a constant, an error will be output.

This function must not be executed when the RX processor mode is user mode. If executed, a privileged instruction exception of the RX occurs due to the specifications of the MVTIPL instruction.

unsigned char get_ipl(void)

```
[Description]
Refers to the interrupt mask level.
[Header]
<machine.h>
[Parameters]
-
[Return value]
Interrupt mask level
[Example]
#include <machine.h>
extern unsigned char level;
void main(void)
{
    level=get_ipl();// Obtains the PSW.IPL value and
    // stores it in level.
}
```

[Remarks] If a value smaller than 0 or greater than 7 is specified as level, an error will be output.

```
void set_psw(unsigned long data)
```

[Description] Sets a value to PSW. [Header] <machine.h> [Parameters] dataValue to be set [Return value]

[Example]

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
    set_psw(data);// Sets PSW to the value specified by data.
}
```

[Remarks]

Due to the specifications of the RX instruction set, a write to the **PM** bit of **PSW** is ignored. In addition, a write to **PSW** is ignored when the RX processor mode is user mode.

unsigned long get_psw(void)

[Description] Refers to the PSW value.



[Header] <machine.h> [Parameters]

[Return value] PSW value [Example]

#include <machine.h>
extern unsigned long ret;
void main(void)
{
 ret=get_psw();// Obtains the PSW value and stores it in ret.

}

[Remarks]

In some cases, the timing at which the **PSW** value is obtained differs from the timing at which **get_psw** was called, due to the effect of optimization. Therefore when a code using the **C**, **Z**, **S**, or **O** flag included in the return value of this function is written after some sort of operation, correct operation will not be guaranteed.

void set_fpsw(unsigned long data)

```
[Description]
Sets a value to FPSW.
[Header]
<machine.h>
[Parameters]
dataValue to be set
[Return value]
-
[Example]
#include <machine.h>
extern unsigned long data;
void main(void)
{
set_fpsw(data);// Sets FPSW to the value specified by data.
}
```

unsigned long get_fpsw(void)

```
[Description]
Refers to the FPSW value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
FPSW value
[Example]
#include <machine.h>
extern unsigned long ret;
void main(void)
{
ret=get_fpsw();// Obtains the FPSW value and stores it
// in ret.
}
```

[Remarks]

In some cases, the timing at which the **FPSW** value is obtained differs from the timing at which **get_fpsw** was called, due to the effect of optimization. Therefore when a code using the CV, CO, CZ, CU, CX, CE, FV, FO, FZ, FU, FX, or FS



flag included in the return value of this function is written after some sort of operation, correct operation will not be guaranteed.

void set_usp(void * data)

```
[Description]
Sets a value to USP.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]
#include <machine.h>
extern void * data;
void main(void)
{
set_usp(data);// Sets USP to the value specified by data.
}
```

void * get_usp(void)

```
[Description]
Refers to the USP value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
USP value
[Example]
#include <machine.h>
extern void * ret;
void main(void)
{
ret=get_usp();// Obtains the USP value and stores it in ret.
}
```

void set_isp(void * data)

```
[Description]
Sets a value to ISP.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]
#include <machine.h>
extern void * data;
void main(void)
{
set_isp(data);// Sets ISP to the value specified by data.
}
```

[Remarks]



Due to the specifications of the **MVTC** instruction used in this function, a write to **ISP** is ignored when the RX processor mode is user mode.

void * get_isp(void)	
----------------------	--

[Description] Refers to the ISP value. [Header] <machine.h> [Parameters] -[Return value] ISP value [Example] #include <machine.h> extern void * ret; void main(void) { ret=get_isp();// Obtains the ISP value and stores it in ret. }

void set_intb (void * data)

[Description] Sets a value to INTB. [Header] <machine.h> [Parameters] dataValue to be set [Return value]

[Example]

```
#include <machine.h>
extern void * data;
void main(void)
{
    set_intb (data);// Sets INTB to the value specified by data.
}
```

-

[Remarks] Due to the specifications of the **MVTC** instruction used in this function, a write to **INTB** is ignored when the RX processor mode is user mode.

void * get_intb(void)

[Description] Refers to the INTB value. [Header] <machine.h> [Parameters]

[Return value] INTB value [Example]



```
#include <machine.h>
extern void * ret;
void main(void)
{
    ret=get_intb();// Obtains the INTB value and stores it in ret.
}
```

void set_bpsw(unsigned long data)

```
[Description]
Sets a value to BPSW.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
-
[Example]
#include <machine.h>
```

```
extern unsigned long data;
void main(void)
{
set_bpsw (data);// Sets BPSW to the value specified by data.
```

```
}
```

```
[Remarks]
```

Due to the specifications of the **MVTC** instruction used in this function, a write to **BPSW** is ignored when the RX processor mode is user mode.

```
unsigned long get_bpsw(void)
```

```
[Description]
Refers to the BPSW value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
BPSW value
[Example]
#include <machine.h>
extern unsigned long ret;
void main(void)
{
ret=get_bpsw ();// Obtains the BPSW value and stores it
// in ret.
}
```

void set_bpc(void * data)

[Description] Sets a value to BPC. [Header] <machine.h> [Parameters] data Value to be set [Return value]

[Example]



```
#include <machine.h>
extern void * data;
void main(void)
{
    set_bpc(data);// Sets BPC to the value specified by data.
}
```

[Remarks]

Due to the specifications of the **MVTC** instruction used in this function, a write to **BPC** is ignored when the RX processor mode is user mode.

void * get_bpc(void)
Description]
Refers to the BPC value.
Header]
<pre><machine.h></machine.h></pre>
Parameters
Return value] 3PC value Example]
#include <machine.h> extern void * ret; void main(void) {</machine.h>
ret=get_bpc();// Obtains the BPC value and stores it in ret.
}

```
void set_fintv(void * data)
```

```
[Description]
Sets a value to FINTV.
[Header]
<machine.h>
[Parameters]
data Value to be set
[Return value]
```

[Example]

```
#include <machine.h>
extern void * data;
void main(void)
{
    set_fintv(data);// Sets FINTV to the value specified by data.
}
```

[Remarks]

Due to the specifications of the **MVTC** instruction used in this function, a write to **FINTV** is ignored when the RX processor mode is user mode.

void * get_fintv(void)

[Description] Refers to the FINTV value. [Header] <machine.h> [Parameters]

[Return value]



```
FINTV value
[Example]
```

```
#include <machine.h>
extern void * ret;
void main(void)
{
    ret=get_fintv();// Obtains the FINTV value and stores it
        // in ret.
}
```

signed long long emul(signed long data1, signed long data2)

[Description] Performs signed multiplication of significant 64 bits. [Header] <machine.h> [Parameters] data 1 Input value 1 data 2 Input value 2 [Return value] Result of signed multiplication (signed 64-bit value) [Example]

```
unsigned long long emulu(unsigned long data1, unsigned long data2)
```

```
[Description]
Performs unsigned multiplication of significant 64 bits.
[Header]
<machine.h>
[Parameters]
data 1 Input value 1
data 2 Input value 2
[Return value]
Result of unsigned multiplication (unsigned 64-bit value)
[Example]
 #include <machine.h>
 extern unsigned long long ret;
 extern unsigned long data1, data2;
 void main(void)
 {
      ret=emulu(data1, data2);// Calculates the value of
          // "data1 * data2" and stores it in ret.
 }
```

```
void chg_pmusr(void)
```

[Description] Switches the RX processor mode to user mode. [Header] <machine.h>



[Parameters]

[Return value]

[Example]

```
#include <machine.h>
void main(void);
void Do_Main_on_UserMode(void)
{
    chg_pmusr();// Switches the RX processor mode to user mode.
    main();// Executes the main function.
}
```

[Remarks]

This function is provided for a reset processing function or interrupt function. Usage in any other function is not recommended.

The processor mode is not switched when the RX processor mode is user mode.

Since the stack is switched from the interrupt stack to the user stack when this function is executed, the following conditions must be met in a function that is calling this function. If the conditions are not met, code does not operate correctly because the stack is not the same before and after this function has been executed.

- Execution cannot be returned to the calling function.
- The auto variable cannot be declared.

- Parameters cannot be declared.

void set_acc(signed long long data)

```
[Description]
Sets a value to ACC.
[Header]
<machine.h>
[Parameters]
data Value to be set to ACC
[Return value]
-
[Example]
#include <machine.h>
void main(void)
{
signed long long data = 0x123456789ab0000LL;
```

```
set_acc(data);// Sets ACC to the value specified by data.
```

```
}
```

signed long long get_acc(void)

```
[Description]
Refers to the ACC value.
[Header]
<machine.h>
[Parameters]
-
[Return value]
ACC value
[Example]
```



```
/* Example of program using the function to save and restore ACC by*/
/* get_acc and set_acc*/
#include <machine.h>
signed long a, b, c;
void func(void)
{
    signed long long bak_acc = get_acc();
    // Obtains the ACC value and saves it
    // in bak_acc.
    c = a * b;// Multiplication (ACC is damaged).
    set_acc(bak_acc);// Restores ACC with a value saved by
    // bak_acc.
}
```

[Remarks]

Due to the specifications of the RX instruction set, contents in the lower 16 bits of **ACC** cannot be obtained. This function returns the value of 0 for these bits.

```
void setpsw_i(void)
```

```
[Description]
Sets the interrupt enable bit (I bit) in PSW to 1.
[Header]
<machine.h>
[Parameters]
-
[Return value]
-
[Example]
#include <machine.h>
void main(void)
{
setpsw_i();// Sets the interrupt enable bit to 1.
}
```

[Remarks]

Due to the specifications of the SETPSW instruction used by this function, writing to the interrupt enable bit is ignored when the RX processor mode is set to user mode.

void clrpsw_i(void)

```
[Description]
Clears the interrupt enable bit (I bit) in PSW to 0.
[Header]
<machine.h>
[Parameters]
-
[Return value]
```

-

[Example]

```
#include <machine.h>
void main(void)
{
    clrpsw_i();// Clears the interrupt enable bit to 0.
}
```

[Remarks]

Due to the specifications of the CLRPSW instruction used by this function, writing to the interrupt enable bit is ignored when the RX processor mode is set to user mode.



long macl(short * data1, short * data2, unsigned long count)

[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]
        Start address of values 1 to be multiplied
data1
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 specifica-

tions #pragma interrupt.

short macw1(short* data1, short* data2, unsigned long count) short macw2(short* data1, short* data2, unsigned long count)

[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.**

```
void set_extb (void * data)
```

```
[Description]
Sets a value for EXTB.
[Header]
<machine.h>
[Parameters]
dataValue to be set
[Return value]
```

[Example]

```
#include <machine.h>
extern void * data;
void main(void)
{
    set_extb (data);// Sets EXTB to the value specified by data.
}
```

[Remarks]

This function is only usable when rxv2 is specified for the isa option or the environment variable ISA_RX. In other cases, this option will lead to an error at the time of compilation.

Due to the specifications of the **MVTC** instruction used in this function, a write to **EXTB** is ignored when the RX processor mode is user mode.

void * get_extb(void)

[Description] Refers to the EXTB value. [Header] <machine.h>



```
[Parameters]
```

```
[Return value]
[EXTB value
[Example]
#include <machine.h>
extern void * ret;
void main(void)
{
ret=get_extb();// Obtains the EXTB value and stores it in ret.
}
```

[Remarks]

This function is only usable when rxv2 is specified for the isa option or the environment variable ISA_RX. In other cases, this option will lead to an error at the time of compilation.

3.2.7 Section Address Operators

Table 3.26 lists the section address operators.

Table 3.25	Section Addres	s Operators
------------	----------------	-------------

No.	Section Address Operator	Description
1	sectop(" <section name="">")</section>	Refers to the start address of the specified <section name="">.</section>
2	secend(" <section name="">")</section>	Refers to the end address + 1 of the specified <section name="">.</section>
3	secsize(" <section name="">")</section>	Generates the size of the specified <section name="">.</section>

sectop(" <section name="">")</section>		
<pre>secend("<section name="">")</section></pre>		
secsize(" <section name="">")</section>		

[Description]

- ___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) <u>sectop</u>, <u>secend</u>



__sectop refers to the start address of the specified <section name>.

```
#include <machine.h>
 #pragma section $DSEC
 static const struct {
   void *rom s; /* Start address of the initialized data section in ROM */
   void *rom e; /* End address of the initialized data section in ROM */
   void *ram_s; /* Start address of the initialized data section in RAM */
 } DTBL[]={__sectop("D"), __secend("D"), __sectop("R")};
 #pragma section $BSEC
 static const struct {
   void *b s; /* Start address of the uninitialized data section */
   void *b e; /* End address of the uninitialized data section */
 } BTBL[]={_______sectop("B"), _____secend("B")};
 #pragma section
 #pragma stacksize si=0x100
 #pragma entry INIT
 void main(void);
 void INIT(void)
 {
    INITSCT();
   main();
   sleep();
 }
(2) secsize
 /* size of section B */
```

unsigned int size_of_B = __secsize("B");

[Remarks]

In an application that enables the PIC/PID function, **___sectop** and **___secend** is processed as the addresses determined at linkage.

For details of the PIC/PID function, refer to the descriptions of the pic and pid options in Usage of PIC/PID Function.

3.3 Modification of C Source

By using expanded function object with high efficiency can be created.

Here, two methods are described for shifting to the CCRX from other C compiler and shifting to C compiler from the CCRX.

<From other C compiler to the CCRX>

- #pragma

C source needs to be modified, when C compiler supports the #pragma. Modification methods are examined according to the C compiler specifications.

- Expanded Specifications

It should be modified when other C compilers are expanding the specifications such as adding keywords etc. Modified methods are examined according to the C compiler specifications.

Note #pragma is one of the pre-processing directives supported by ANSI. The character string next to #pragma is made to be recognized as directives to C compiler. If that directive does not supported by the compiler, #pragma directive is ignored and the compiler continues the process and ends normally.

<From the CCRX to other C compiler>

- The CCRX, either deletes key word or divides # fdef in order shift to other C compiler as key word has been added as expanded function.

Examples 1. Disable the keywords

#ifndefRX	
#define interrupt #endif	/*Considered interrupt function as normal function*/



Examples 2. Change to other type

#ifdef __RX

#define bit char /*Change bit type variable to char type variable*/ #endif

3.4 Function Calling Interface

This section describes how to handle arguments when a program is called by the CCRX.

- Rules Concerning the Stack
- Rules Concerning Registers
- Rules Concerning Setting and Referencing Parameters
- Rules Concerning Setting and Referencing Return Values
- Examples of Parameter Allocation

3.4.1 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 3.2 illustrates the stack frame status immediately after a function call.

Figure 3.2 Allocation and Deallocation of a Stack Frame

At a function call and immediately after control returns from a function



3.4.2 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 3.27 shows the rules for using registers.

Table 3.26	Rules to Use Registers
------------	------------------------

Register	Register Value Does Not Change During Function Call	Function Entry	Function Exit	High-Speed Interrupt Register ^{*1}	Base Reg- ister* ²	PID Regis- ter*3
R0	Guaranteed	Stack pointer	Stack pointer	—		
R1	Not guaranteed	Parameter 1	Return value 1	_		
R2	Not guaranteed	Parameter 2	Return value 2	—		



Register	Register Value Does Not Change During Function Call	Function Entry	Function Exit	High-Speed Interrupt Register ^{*1}	Base Reg- ister* ²	PID Regis- ter*
R3	Not guaranteed	Parameter 3	Return value 3	_		
R4	Not guaranteed	Parameter 4	Return value 4			
R5	Not guaranteed		(Undefined)	_	_	
R6	Guaranteed	—	(Value at func- tion entry is held)			
R7	Guaranteed	—	(Value at func- tion entry is held)			
R8	Guaranteed	—	(Value at func- tion entry is held)		0	
R9	Guaranteed		(Value at func- tion entry is held)		0	0
R10	Guaranteed	—	(Value at func- tion entry is held)	0	0	0
R11	Guaranteed		(Value at func- tion entry is held)	0	0	0
R12	Guaranteed		(Value at func- tion entry is held)	0	0	0
R13	Guaranteed		(Value at func- tion entry is held)	0	0	0
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		ter. *4			
PC		Program counter*5	i			
PSW	Not guaranteed		(Undefined)			
FPSW	Not guaranteed		(Undefined)			
ACC	Not guaranteed* ⁶		(Undefined) * ⁶			
INTB BPC BPSW FINTV CPEN		No change ^{*4}				

Notes 1. The high-speed interrupt function may use some or all four registers among R10 to R13, depending on the **fint_register** option. Registers assigned to the high-speed interrupt function cannot be used for other purposes. For details on the function, refer to the description on the option.

Notes 2. The base register function may use some or all six registers among R8 to R13, depending on the **base** option. Registers assigned to the base register function cannot be used for other purposes. For details on the function, refer to the description on the option.

Notes 3. The PID function may use one of R9 to R13, depending on the **pid** option. The register assigned to the PID function cannot be used for other purposes. For details on the function, refer to the description on the option.

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

3.4.3 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
 - (a) Parameters whose types are declared by a prototype declaration are converted to the declared types.
 - (b) Parameters whose types are not declared by a prototype declaration are converted according to the following rules.
 - int type of 2 bytes or less is converted to a 4-byte int type.
 - float type parameters are converted to **double** type parameters.
 - Types other than the above are not converted.

Example



(3) Parameter Area Allocation

Parameters are allocated to registers or to a parameter area on the stack. Figure 3.3 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 3.28 lists general rules on parameter area allocation.







Param	eters Allocated to Regi	sters	Parameters Allocated to	
Target Type	Parameter Storage Registers	Allocation Method	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 refer- ence	One register among R1 to R4	Sign extension is per- formed for signed char or (signed) short type, and zero extension is per- formed for (unsigned) char type, and the results are allocated. All other types are allo- cated without any exten- sion performed.	 Parameters whose types are other than target types for register passing Parameters of a func- tion which has been declared by a prototype declaration to have vari- able-number parameters*³ When the number of registers not yet allocated 	
(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 allo- cated to the larger num- bered register.	with parameters among R1 to R4 is smaller than the number of registers needed to allocate param- eters	
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, parame- ters are allocated in 4-byte units to the registers start- ing with the smallest num- bered register.		

Table 3.27 General Rules on Parameter Area Allocation	Table 3.27	General Rules on Parameter Area Allocation
---	------------	--

Notes 1. When **dbl_size=8** is not specified.

- Notes 2. When **dbl_size=8** is specified.
- Notes 3. If a function has been declared to have variable parameters by a prototype declaration, parameters which do not have a corresponding type in the declaration and the immediately preceding parameter are allocated to the stack. For parameters which do not have a corresponding type, an integer of 2 bytes or less is converted to **long** type and **float** type is converted to **double** type so that all parameters will be handled with a boundary alignment number of 4.

Example

int f2(int,int,int,);
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 3.28 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**.

3.4.4 Rules Concerning Setting and Referencing Return Values

General rules concerning return values and the areas for setting return values are described.

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

Table 3.28	Return Value Type and Set	ting Area
------------	---------------------------	-----------

Return Value Type	Return Value Setting Area
signed char, (unsigned) char, (signed) short, unsigned short, (signed) int, unsigned int, (signed) long, unsigned long, float, double* ² , long double* ² , pointer, bool, _Bool, reference, and pointer to a data member	R1 Note however that the result of sign extension is set for signed char or (signed) short type, and the result of zero extension is set for (unsigned) char or unsigned short type.
double ^{*3} , long double ^{*3} , (signed) long long, and unsigned long long	R1, R2 The lower four bytes are set to R1 and the upper four bytes are set to R2.
Structure, union, or class whose size is 16 bytes or less and is also a multiple of 4	They are set from the beginning of the memory image in 4- byte units in the order of R1, R2, R3, and R4.
Structure, union, or class other than those above	Return value setting area (memory)* ¹

- Notes 1. When a function return value is to be written to memory, the return value is written to the area indicated by the return value address. The caller must allocate the return value setting area in addition to the parameter area, and must set the address of the return value setting area in R15 before calling the function.
- Notes 2. When dbl_size=8 is not specified.
- Notes 3. When **dbl_size=8** is specified.

3.4.5 Examples of Parameter Allocation

Examples of parameter allocation are shown in the following. Note that addresses increase from the right side to the left side in all figures (upper address is on the left side).

Examples 1. Parameters matching the type to be passed to registers are allocated, in the order in which they are declared, to registers R1 to R4.

If there is a parameter that will not be allocated to registers midway, parameters after that will be allocated to registers. The parameter will be placed on the stack at an address corrected to match the boundary alignment number of that parameter.



int f(<registe< th=""><th>er\$></th><th></th><th></th></registe<>	er\$>			
unsigned char,	[
long long,	R1	0x000000(Zero ex	tension)	0x01	
long long,	R2	0x0000	00002		
short,	R3	0x0000000			
int,		0x0000			
char,	R4	(Sign extension)	0x0004		
short,					
char,	<stack></stack>				
char,	*(R0+0)	0x0000003			
short);	*(R0+4)	0x0000000			
	*(R0+8)	0x0000005			
I(1,2,3,4,5,6,7,8,9,10); /*	*(R0+12)	0x0007	Empty area	0x06	
** 1, 2, and 4 are allocated to registers */	*(R0+16)	0x000A	0x09	0x08	

Examples 2. Parameters of a structure or union whose size is 16 bytes or less and is also a multiple of 4 are allocated to registers. Parameters of all other structures and unions are allocated to the stack.

	<register< th=""><th>'s ></th><th></th><th></th><th></th></register<>	's >			
<pre>union U (int a[2]; int b;) u; struct S (short d; char c[4];) s; struct T (char d; char f[2]; char e;) t;</pre>	R1 R2	u.a[0] (=u.b) u.a[1]			
int f(union U, struct S, struct T);	R3	t.e	t.f[1]	t.f[0]	t.g
: f(u, s, t);					
r•	<stack></stack>				
** u is allocated to a register because it is 8 bytes ** s is allocated to the stack because it is 6 bytes	*(R0+0)	s.c[1]	s.c[0]	S	d
** t is allocated to a register because it is 4 bytes	*(R0+4)	Empty	area	s.c[3]	s.c[2]

Examples 3. When declared in a prototype declaration as a function with a variable-number of parameters, the parameters without corresponding types and the immediately preceding parameter are allocated to the stack in the order in which they are declared.

<registers></registers>				
<pre>int f(int, float, int, int,) .</pre>	R1	0x0000000		
f(0, 1.0, 2, 3, 4)	R2	0x3F800000		
		0x0000002		
<5 *(Stack>	0x0000003		
*(R0+4)	0x0000004		

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

	<registe< td=""><td>rs></td></registe<>	rs>
struct S{char a[7];};	R1	0x0000001
struct S f (
int al,	R2	0x0000002
int a2,	R3	0x0000003
int a3,	R4	0x0000004
int a4,	L	
int a5);	R15	Return value address(= R0 + 4)
:	L	
f(1,2,3,4,5);	<stack></stack>	
	*(R0+0)	0x0000005
	*(R0+4)	Return value setting area
	*(R0+8)	Empty area

Examples 5. When setting the return value to memory, normally a stack is allocated, as shown in example 4. In the case of setting the return value to a variable, however, no stack is allocated and it is directly set to the memory area for that variable. In this case, the address for the variable is set to R15.



3.5 List of Section Names

This section describes the sections for CCRX.

Each of the regions for execution instructions and data of the relocatable files output by the assembler comprises a section. A section is the smallest unit for data placement in memory. Sections have the following properties.

- Section attributes

code Stores execution instructions

data Stores data that can be changed

romdata Stores fixed data

- Format type

Relative-address format: A section that can be relocated by the optimizing linkage editor.

Absolute-address format: A section of which the address has been determined; it cannot be relocated by the optimizing linkage editor.

Initial values



Specifies whether there are initial values at the start of program execution. Data which has initial values and data which does not have initial values cannot be included in the same section. 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.

3.5.1 C/C++ Program Sections

The correspondence between memory areas and sections for C/C++ programs and the standard library is described in table 3.30.

Name	Section		Format	Initial Value	Align-	Description
	Name	Attribute	турс	Write Opera- tion	Number	
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}	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}	romdata	Relative	Yes Possible	2 bytes	
	D_1* ¹ * ² * ⁶	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	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 process- ing/ postprocess- ing data area	C\$INIT	romdata	Relative	Yes Not possible	4 bytes	Stores addresses of con- structors and destructors called for global class objects

 Table 3.29
 Summary of Memory Area Types and Their Properties



Name	Section		Format	Initial Value	Align-	Description
	Name	Attribute	Туре	Write Opera- tion	Number	
C++ virtual function table area	C\$VTBL	romdata	Relative	Yes Not possible	4 bytes	Stores data for calling the vir- tual 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
Heap area			Relative	No Possible		Area used by library func- tions malloc, realloc, cal- loc, and new
Absolute address variable area	\$ADDR_ <section>_ <address> *³</address></section>	data	Absolute	Yes/No Possible/ Not possi- ble* ⁴		Stores variables specified by #pragma address
Variable vector area	C\$VECT	romdata	Relative	No Possible	4 bytes	Variable vector table
Literal area	L*2	romdata	Relative	Yes Possible/ Not possible	4 bytes	Stores string literals and ini- tializers used for dynamic ini- tialization of aggregates

Notes 1. Section names can be switched using the section option.

- Notes 2. Specifying a section with a boundary alignment of 4 when switching the section names also changes the section name of sections with a boundary alignment of 1 or 2.
- Notes 3. <section> is a C, D, or B section name, and <address> is an absolute address (hexadecimal).
- Notes 4. The initial value and write operation depend on the attribute of <section>.
- Notes 5. The section name can be changed by using the section option. In this case, the C section can be selected as the changed name.
- Notes 6. The section name can be changed through #pragma section.
- Notes 7. Specifying the instalign4 or instalign8 option, #pragma instalign4, or #pragma instalign8 changes the boundary alignment to 4 or 8.
- Notes 8. If an endian not matching the endian option has been specified in #pragma endian, a dedicated section is created to store the relevant data. At the end of the section name, _B is added for #pragma endian big, and _L is added for #pragma endian little.
- Examples 1. A program example is used to demonstrate the correspondence between a C program and the compilergenerated sections.

-	
}	
void main () $\{$	
const short $c = 0$;
char b;	
int a=1;	

C program

Program area (main(){})	Р
Constant area (c)	C_2
Initialized data area (a)	D
Uninitialized data area (b)	B_1
Areas generated by the compiler and stored data	Section name


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



3.5.2 Assembly Program Sections

In assembly programs, the **.SECTION** control directive is used to begin sections and declare their attributes, and the **.ORG** control directive is used to declare the format types of sections.

For details on the control directives, refer to section 10.3, Assembler Directive Coding.

Example An example of an assembly program section declaration is shown below.

	.SECTIO	DNA,CODE,ALIGN=4;(1)
START:	MOV.L	#CONST,R4
I OOP	MOV.L ADD MOV.L MOV.L	[R4],R5 #10,R5,R3 #100,R4 #ARRAY,R5
	MOV.L SUB CMP BNE	R3,[R5+] #1,R4 #0,R4 LOOP
EXIT:	RTS	
;	.SECTIO .ORG .glb	DNB,ROMDATA;(2) 02000H CONST
	.LWORI	D05H
, BASE	.SECTIO	DNC,DATA,ALIGN=4;(3) BASE
DAUL.	.blkl .END	100

(1) Declares a **code** section with section name **A**, boundary alignment 4, and relative address format.

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

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



3.5.3 Linking Sections

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

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



(2) Sections with the same name but different boundary alignments are linked after alignment. Section alignment uses the larger of the section alignments.



(3) When sections with the same name include both absolute-address and relative-address formats, relativeaddress 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







4. ASSEMBLY LANGUAGE SPECIFICATIONS

This chapter describes the assembly language specifications supported by the RX assembler.

4.1 Description of Source

This section explains description of source, expressions, and operators.

4.1.1 Description

The following shows the mnemonic line format. [label][operation[∆operand(s)]][comment] Coding example:

MOV.L

LABEL1	:
	_

[R1], R2 Example of a mnemonic.

Operands

Comment

Label

(1) Label

Define a name for the address of the mnemonic line.

(2) Operation

Write a mnemonic or a directive.

Operand(s) (3)

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.

Operation

4.1.2 Names

Table 4.1

Desired names can be defined and used in assembly-language files. Names are classified into the following types.

Туре	

Types of Name

Туре	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.

Flag names (U, I, O, S, Z, and C), which are reserved words, can be used only for section names. Note

4.1.3 Coding of Labels

Be sure to append a colon (:) to the end of a label.

Example

LABEL1:

Defining a symbol name which is the same as that of an existing section is not possible. If a section and symbol with the same name are defined, the section name will be effective, but the symbol name will lead to an A2118 error.

4.1.4 Coding of Operation

- Format

mnemonic [size specifier (branch distance specifier)]

- Description
 - An instruction consists of the following two elements.
- (1) Mnemonic: Specifies the operation of the instruction.
- (2) Size specifier: Specifies the size of the data which undergoes the operation.
- (1) Mnemonic

A mnemonic specifies the operation of the instruction.

Example:

MOV: Transfer instruction ADD: Arithmetic instruction (addition instruction)

(2) Size Specifier

A size specifier specifies the size of the operand(s) in the instruction code.

- Format

.size

- Description

A size specifier specifies the operation size of the operand(s). More exactly, it specifies the size of data to be read to execute the instruction. The following can be specified as **size**.

Table 4.2	Size Specifiers
-----------	-----------------

size	Description
В	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 4.3	Branch Distance Specifiers
-----------	----------------------------

length	Description	
S	3-bit PC forward relative	(+3 to +10)
В	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:

 $\texttt{BRA.W} \quad \texttt{label} \ \dots \ \textbf{Specifies 16-bit relative}.$

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.

Instruction		.S	.В	.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

Table 4.4 Branch Distance Specifiers for Each Branch Instruction

4.1.5 Coding of Operands

(1) Numeric Value

Five types of numeric values described below can be written in programs. The written values are handled as 32-bit signed values (except floating-point values).

(a) Binary Number

Use digits 0 and 1, and append B or b as a suffix.

- Examples

1011000B 1011000b

(b) Octal Number

Use digits 0 to 7, and append O or o as a suffix.

- Examples

607020 607020



(c) Decimal Number Use digits 0 to 9.

- Example

9243

(d) Hexadecimal Number

Use digits 0 to 9 and letters A to F and a to f, and append H or h as a suffix. When starting with a letter, append 0 as a prefix.

- Examples

0A5FH 5FH 0a5fh 5fh

(e) Floating-Point Number

A floating-point number can be written only as the operand of the **.FLOAT** or **.DOUBLE** directive. No floating-point number can be used in expressions. The following range of values can be written as floating-point numbers. **FLOAT** (32 bits):1.17549435 × 10^{-38} to 3.40282347 × 10^{-38} **DOUBLE** (64 bits):2.2250738585072014 × 10^{-308} to 1.7976931348623157 × 10^{-308}

- Format

(mantissa)E(exponent)
(mantissa)e(exponent)

- Examples

3.4E35	;3.4×10**35
3.4e-35	;3.4×10**-35
5E20	;-0.5×10**20
5e-20	;5.0×10**-20

4.1.6 Expression

A combination of numeric values, symbols, and operators can be written as an expression.

- A space character or a tab can be inserted between an operator and a numeric value.
- Multiple operators can be used in combination.
- When using an expression as a symbol value, make sure that the value of the expression is determined at assembly.
- A character constant must not be used as a term of an expression.
- The expression value as a result of operation must be within the range from –2147483648 to 2147483647. The assembler does not check if the result is outside this range.

(a) Operator

The following is a list of the operators that can be written in programs.

- Unary operators

Table 4.5	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 4.6 Binary Operators

Operator	Function
+	Adds the lvalue and rvalue.
-	Subtracts the rvalue from the lvalue.
*	Multiplies the lvalue and rvalue.
1	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 Ivalue 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 4.7Conditional 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 4.8 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.

Precedence	Operator Type	Operator
1	Precedence designation operator	0
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	>, <, >=, <=, ==, !=

Table 4.9 Order of Expression Evaluation

(1) 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 FFFFFFFh.

[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 FFFFFFFh. **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 4.10	Scaling Rule	es 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 speci- fier (.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 FFFFFFFh. 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 FFFFFFFh. 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.

 $[-\mbox{Rn}]$ (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 FFFFFFFh. 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 FFFFFFFh. 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 $3 \le pcdsp: 3 \le 10$.

The effective address range is 00000000h to FFFFFFFh. 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**: $-128 \le pcdsp$: $8 \le +127$

- For **.W**: −32768 ≤ pcdsp:16 ≤ +32767
- For **.A**: -8388608 ≤ pcdsp:24 ≤ +8388607

The effective address range is 00000000h to FFFFFFFh.

(2) 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

(3) 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 ${\bf B}$ for bit manipulation instructions or assumes ${\bf L}$ for other instructions.

The following shows available size extension specifiers and their function.

Table 4.11	Size Extension	Specifiers
------------	----------------	------------

Size Extension Specifier	Function
В	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

4.1.7 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.



4.1.8 Selection of Optimum Instruction Format

Some of the RX Family microcontroller instructions provide multiple instruction formats for an identical single processing.

The assembler selects the optimum instruction format that generates the shortest code according to the instruction and addressing mode specifications.

(1) Immediate Value

For an instruction having an immediate value as an operand, the assembler selects the optimum one of the available addressing modes according to the range of the immediate value specified as the operand. The following shows the immediate value ranges in the order of priority.

#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 7FFFFH
#imm:32	-2147483648 to 4294967295	-80000000H to 0FFFFFFFH

Table 4.12	Ranges of Immediate Values
------------	----------------------------

Notes 1. Hexadecimal values can also be written in 32 bits.

Example: Decimal "-127" = hexadecimal "-7FH" can be written as "0FFFFF81H".

Notes 2. The **#imm** range for **src** in the **INT** instruction is 0 to 255.

Notes 3. The **#imm** range for **src** in the **RTSD** instruction is four times the **#uimm:8** range.

(2) ADC and SBB Instructions

The following shows the **ADC** and **SBB** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Note The following table does not show the instruction formats and operands for which code selection is not optimized. When the processing size is not shown in the table, **L** is assumed.

Instruction Format	Target of 0	Code Size		
	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 4.14	Instruction Formats	s of ADD Instruction

Instruction Format	Target of Optimum Selection			Code Size
	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 4.15	Instruction Formats of AND,	OR, S	SUB,	and MUL	Instructions
------------	-----------------------------	-------	------	---------	--------------

Instruction Format	Target of Optimum Selection			Code Size	
	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 \neq UB)	
	dsp:16[Rs].memex	_	Rd	4 (memex = UB), 5 (memex \neq 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 4.16 Instruction Formats of BMCnd Instruction

Instruction Format	Processing	Target of Optimum Selection			Code Size
	Size	src	src2	dest	[Bytes]



BMCnd src,dest	В	#imm:3	—	dsp:8[Rs].B	4
	В	#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 4 17	Instruction	Formats	of CMP	Instruction
	Instruction	Furnals		monuction

Instruction Format	Processing	Target of Optim	um Selectio	n	Code Size
	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 4.18 Instruction F	ormats of DIV, DIVU	, EMUL,	EMULU, ITOF,	MAX, MIN,	TST, a	and XOR I	nstructions
--------------------------	---------------------	---------	--------------	-----------	--------	-----------	-------------

Instruction Format	Target of Optimur	n Selectio	Code Size	
	SIC	src2	dest	[Bytes]
	#simm:8	_	Rd	4
MAX/MIN/TST/XOR	#simm:16	_	Rd	5
	#simm:24	_	Rd	6
	#imm:32	_	Rd	7
src,dest	dsp:8[Rs].memex	_	Rd	4 (memex = UB), 5 (memex ≠ UB)
	dsp:16[Rs].memex		Rd	5 (memex = UB), 6 (memex ≠ UB)

In the ITOF instruction, #simm:8/16/24 and #imm:32 are not allowed for src.

(8) FADD, FCMP, FDIV, FMUL, and FTOI Instructions

The following shows the **FADD**, **FCMP**, **FDIV**, **FMUL**, and **FTOI** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

 Table 4.19
 Instruction Formats of FADD, FCMP, FDIV, FMUL, and FTOI Instructions

Instruction Format	Target of O	Code Size		
	SrC	src2	dest	[Bytes]



FADD/FCMP/FDIV/	#imm:32	_	Rd	7
src,dest	dsp:8[Rs].L		Rd	4
	dsp:16[Rs].L		Rd	5

In the FTOI instruction, #imm:32 is not allowed for src.

(9) MVTC, STNZ, and STZ Instructions

The following shows the **MVTC**, **STNZ**, and **STZ** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 4 20	Instruction Formats	of MVTC STN7	and STZ Instructions

Instruction Format	Target of O	on	Code Size	
	src	src2	dest	[Bytes]
MVTC/STNZ/STZ	#simm:8	_	Rd	4
SIC, 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 4.21	Instruction	Formats of	MOV	Instruction
------------	-------------	------------	-----	-------------

Instruction	size	Processing	Target of	Optimum	Selection	Code
Format		5120	src	src2	dest	[Bytes]
MOV(.size)	B/W/L	size	Rs (Rs=R0-R7)		dsp:5[Rd] (Rd=R0-R7)	2
src,aest	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
	В	В	#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
	В	В	#imm:8	—	dsp:8[Rd]	4
	W/L	W/L	#simm:8	_	dsp:8[Rd]	4



Instruction	size	Processing	Target of	Code		
Format		Size	src	src2	dest	Size [Bytes]
MOV(.size)	W	W	#imm:16	_	dsp:8[Rd]	5
src,dest	L	L	#simm:16	_	dsp:8[Rd]	5
	L	L	#simm:24		dsp:8[Rd]	6
	L	L	#imm:32	_	dsp:8[Rd]	7
	В	В	#imm:8		dsp:16[Rd]	5
	W/L	W/L	#simm:8		dsp:16[Rd]	5
	W	W	#imm:16	_	dsp:16[Rd]	6
	L	L	#simm:16		dsp:16[Rd]	6
	L	L	#simm:24		dsp:16[Rd]	7
	L	L	#imm:32	_	dsp:16[Rd]	8
	B/W/L	L	dsp:8[Rs]		Rd	3
	B/W/L	L	dsp:16[Rs]		Rd	4
	B/W/L	size	Rs	_	dsp:8[Rd]	3
	B/W/L	size	Rs		dsp:16[Rd]	4
	B/W/L	size	[Rs]		dsp:8[Rd]	3
	B/W/L	size	[Rs]	_	dsp:16[Rd]	4
	B/W/L	size	dsp:8[Rs]	_	[Rd]	3
	B/W/L	size	dsp:16[Rs]		[Rd]	4
	B/W/L	size	dsp:8[Rs]		dsp:8[Rd]	4
	B/W/L	size	dsp:8[Rs]	—	dsp:16[Rd]	5
	B/W/L	size	dsp:16[Rs]	—	dsp:8[Rd]	5
	B/W/L	size	dsp:16[Rs]	—	dsp:16[Rd]	6

(11) MOVU Instruction

The following shows the **MOVU** instruction formats and operands for which the assembler selects the optimum code, in the order of selection priority.

Table 4.22	Instruction	Formats	of MOVU	Instruction
------------	-------------	---------	---------	-------------

Instruction	size	Processing	Target of Optimum Selection			Code
Format		Size	src	src2	dest	[Bytes]
MOVU(.size)	B/W	L	dsp:5[Rs] (Rs=R0-R7)	_	Rd (Rd=R0-R7)	2
src,dest	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.

Instruction Format	Target of O	ptimum Selecti	on	Code Size
	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 4.24	Instruction F	ormats o	of ROUND	Instruction
------------	---------------	----------	----------	-------------

Instruction Format	Target of O	Code Size		
	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 4.25	Instruction	Formats	of SCCnd	Instruction
10010 4.20	moulon	i onnato	01000110	moulon

Instruction Format	size	Target of Optimum Selection		Code 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 4.26	Instruction Formats of XCHG Instruction
------------	---

Instruction Format	rmat Processing Size	Target of Optimur	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 4.27 Instruction Formats of BCLR, BNOT, BSET, and BTST Instructions

Instruction Format	Processing	Target of Optimum Selection			Code Size
	5120	src	src2	dest	[Byles]



BCLR/BNOT/BSET/BTST src,dest	В	#imm:3	_	dsp:8[Rd].B	3
	В	#imm:3	—	dsp:16[Rd].B	4
	В	Rs	_	dsp:8[Rd].B	4
	В	Rs	_	dsp:16[Rd].B	5

4.1.9 Selection of Optimum Branch Instruction

- (1) Unconditional Relative Branch (BRA) Instruction
 - (a) Specifiable Branch Distance Specifiers
 - .S 3-bit PC relative (PC + pcdsp:3, $3 \le pcdsp:3 \le 10$)
 - .B 8-bit PC relative (PC + pcdsp:8, $-128 \le pcdsp:8 \le 127$)
 - .W 16-bit PC relative (PC + pcdsp:16, $-32768 \le pcdsp:16 \le 32767$)
 - .A 24-bit PC relative (PC + pcdsp:24, -8388608 ≤ pcdsp:24 ≤ 8388607)
 - .L Register relative (PC + Rs, $-2147483648 \le Rs \le 2147483647$)

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 4.1.4 (3), Branch Distance Specifier.

- When the operand does not satisfy the conditions, the assembler selects the 24-bit PC relative distance (.A).
- (2) Relative Subroutine Branch (BSR) Instruction
 - (a) Specifiable Branch Distance Specifier
 - .W 16-bit PC relative (PC + pcdsp:16, $-32768 \le pcdsp:16 \le 32767$)
 - .A 24-bit PC relative (PC + pcdsp:24, $-8388608 \le pcdsp:24 \le 8388607$)
 - .L Register relative (PC + Rs, $-2147483648 \le Rs \le 2147483647$)

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 4.1.4 (3), Branch Distance Specifier.
 - When the operand does not satisfy the conditions, the assembler selects the 24-bit PC relative distance (.A).
- (3) Conditional Branch (BCnd) Instruction
- (a) Specifiable Branch Distance Specifiers
 - BEQ.S 3-bit PC relative (PC + pcdsp:3, $3 \le pcdsp:3 \le 10$)
 - BNE.S 3-bit PC relative (PC + pcdsp:3, $3 \le pcdsp:3 \le 10$)
 - BCnd.B 8-bit PC relative (PC + pcdsp:8, $-128 \le pcdsp:8 \le 127$)
 - BEQ.W 16-bit PC relative (PC + pcdsp:16, -32768 \leq pcdsp:16 \leq 32767)
 - BNE.W 16-bit PC relative (PC + pcdsp:16, $-32768 \le pcdsp:16 \le 32767$)
- (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

Conditional Branch Instruction	Instruction Replacement	Conditional Branch Instruction	Instruction Replacement
BNC/BLTU dest	BCxx BRA.A dest xx:	BC/BGEU dest	BNCxx BRA.A dest xx:
BLEU dest	BGTUxx BRA.A dest xx:	BGTU dest	BLEUxx BRA.A dest xx:
BNZ/BNE dest	BZxx BRA.A dest xx:	BZ/BEQ dest	BNZxx BRA.A dest xx:
BPZ dest	BNxx BRA.A dest xx:	BO dest	BNOxx BRA.A dest xx:
BGT dest	BLExx BRA.A dest xx:	BLE dest	BGTxx BRA.A dest xx:
BGE dest	BLTxx BRA.A dest xx:	BLT dest	BGExx BRA.A dest xx:

Table 4.28 Replacement Rules of Conditional Branch Instructions

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.

4.1.10 Substitute Register Names (for the PID Function)

The substitute register names listed below can be used instead of the names of general-purpose registers.

Table 4.29 Substitute Register Names

Substitute Register Name	Corresponding General-Purpose Register Name
PID_R0	R0
PID_R1	R1
PID_R2	R2
PID_R3	R3
PID_R4	R4
PID_R5	R5
PID_R6	R6
PID_R7	R7
PID_R8	R8
PID_R9	R9
PID_R10	R10
PID_R11	R11
PID_R12	R12



PID_R13	R13
PID_R14	R14
PID_R15	R15
PID_REG	Register selected as the PID register *1

*1) This indicates the name of the register selected as the PID register when the -pid or -

nouse_pid_register option is specified. For details on the rules for selecting the PID register, refer to the descriptions of the -pid and -nouse_pid_register assembler options.

In assembly-language code that constitutes a master program in which the PID function is enabled, the names of all registers that may be selected as the PID register must be represented by the corresponding substitute register names (rather than the actual names of general-purpose registers such as R13).

When a substitute register name is selected as the PID register, assembling the program with nouse_pid_register enabled will not lead to an error.

[Remark]

Note

Substitute register names are usable even when neither -nouse_pid_register nor the -pid option has been selected.

4.2 Directives

This chapter explains the directives.

Directives are instructions that direct all types of instructions necessary for the assembler.

4.2.1 Outline

Instructions are translated into object codes (machine language) as a result of assembling, but directives are not converted into object codes in principle.

Directives contain the following functions mainly:

- To facilitate description of source programs
- To initialize memory and reserve memory areas

- To provide the information required for assemblers and linkers to perform their intended processing The following table shows the types of directives.

Туре	Directives
Link directives	.SECTION, .GLB, .RVECTOR
Assembler directives	.EQU, .END, .INCLUDE
Address directives	.ORG, .OFFSET, .ENDIAN, .BLKB, .BLKW, .BLKL, .BLKD, .BYTE, .WORD, .LWORD, .FLOAT, .DOUBLE, .ALIGN
Macro directives	.MACRO, .EXITM, .LOCAL, .ENDM, .MREPEAT, .ENDR,MACPARA,MAC-REP, .LEN, .INSTR, .SUBSTR
Specific compiler directives	LINE_TOP,LINE_END, .SWSECTION, .SWMOV, .SWITCH, .INSTALIGN

The following sections explain the details of each directive.

4.2.2 Link Directives

These directives are used for relocatable assembly that enables a program to be written in multiple separate files.

SECTION

This directive declares or restarts a section. [Format]



SECTION∆<section name>

.SECTION∆<section name>,<section attribute> .SECTION∆<section name>,<section attribute>,ALIGN=[2|4|8] .SECTION∆<section name>,ALIGN=[2|4|8] <section attribute>: [CODE|ROMDATA|DATA]

[Description]

This directive declares or restarts a section.

(1) Declaration

This directive defines the beginning of a section with a section name and a section attribute specified.

(2) Restart

This directive specifies restart of a section that has already been declared in the source program. Specify an existing section name to restart it. The section attribute and alignment value declared before are used without change.

The alignment value in the section can be changed through the **ALIGN** specification.

The **.ALIGN** directive can be used in relative-addressing sections defined by the **.SECTION** directive including the **ALIGN** specification or in absolute-addressing sections.

When **ALIGN** is not specified, the boundary alignment value in the section is 1. [Examples]

.SECTIONprogram,CODE NOP .SECTIONram,DATA .BLKB10 .SECTIONtbl1,ROMDATA .BYTE"abcd" .SECTIONtbl2,ROMDATA,ALIGN=8 .LWORD11111111H,22222222H .END

[Remarks]

Be sure to specify a section name.

To use assembler directives that allocate memory areas or store data in memory areas, be sure to define a section through this directive.

To write mnemonics, be sure to define a section through this directive.

A section attribute and ALIGN should be specified after a section name.

A section attribute and ALIGN should be specified with them separated by a comma.

A section attribute and $\ensuremath{\textbf{ALIGN}}$ can be specified in any order.

Select CODE, ROMDATA, or DATA for the section attribute.

The section attribute can be omitted. In this case, the assembler assumes CODE as the section attribute.

When **-endian=big** is specified, only a multiple of 4 can be specified for the start address of an absolute-addressing **CODE** section.

If an absolute-addressing **CODE** section is declared when **-endian=big** is specified, a warning message will be output. In this case, the assembler appends **NOP** (0x03) at the end of the section to adjust the section size to a multiple of 4.

Defining a symbol name which is the same as that of an existing section is not possible. If a section and symbol with the same name are defined, the section name will be effective, but the symbol name will lead to an A2118 error.

The section name \$iop is reserved and cannot be defined. If this is attempted, an A2049 error will be reported.

.GLB

This directive declares that the specified labels and symbols are global.

[Format]

GLB∆<name> .GLB∆<name>[,<name> ?]

[Description]

This directive declares that the specified labels and symbols are global.

When any label or symbol specified through this directive is not defined within the current file, the assembler processes it assuming that it is defined in an external file.

When a label or symbol specified through this directive is defined within the current file, the assembler processes it so that it can be externally referenced.



[Examples]

.GLB name1,name2,name3 .GLB name4 .SECTION program MOV.L #name1,R1

[Remarks]

Be sure to insert a space character or a tab between this directive and the operand. Specify a label name to be a global label as the operand. Specify a symbol name to be a global symbol as the operand. To specify multiple symbol names as operands, separate them by commas (,).

.RVECTOR

This directive registers the specified label or name as a variable vector. [Format]

.RVECTOR∆<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

[Remark]

Be sure to insert a space character or a tab between this directive and the operand.

4.2.3 Assembler Directives

These directives do not generate data corresponding to themselves but controls generation of machine code for instructions. They do not modify addresses.

.EQU

This directive defines a symbol for a 32-bit signed integer value (-2147483648 to 2147483647). [Format]

<name>\Delta.EQU\Delta<numeric value>

[Description]

This directive defines a symbol for a 32-bit signed integer value (-2147483648 to 2147483647). The symbolic debugging function can be used after symbol definition through this directive. [Examples]

symbol .EQU 1 symbol1 .EQU symbol+symbol symbol2 .EQU 2

[Remarks]

The value assigned for a symbol should be determined at assembly.

Be sure to insert a space character or a tab between this directive and the operand.

A symbol can be specified as the operand of symbol definition. Note that forward-reference symbol names must not be specified.



An expression can be specified in the operand.

Symbols can be declared as global.

When this directive and the **.DEFINE** directive declare the same symbol name, the directive to make the declaration first is given priority.

.END

This directive declares the end of an assembly-language file. [Format]

.END

[Description]

This directive declares the end of an assembly-language file.

The source file contents after the line where this directive is written are only output to the source list file; the code corresponding to them is not generated.

[Examples]

.END

[Remarks]

One .END directive should be written in each assembly-language file.

.INCLUDE

This directive inserts the contents of the specified include file to the line where this directive is written in the assemblylanguage file.

[Format]

.INCLUDE∆<include file name>

[Description]

This directive inserts the contents of the specified include file to the line where this directive is written in the assemblylanguage file.

The include file contents are processed together with the contents of the assembly-language file as a single assemblylanguage 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.



4.2.4 Address Directives

These directives control address specifications in the assembler. The assembler handles relocatable address values except for the addresses in absolute-addressing sections.

-	_	_
O	R	G
-		\sim

This directive applies the absolute addressing mode to the section containing this directive.

[Format]

.ORG∆<numeric value>

[Description]

This directive applies the absolute addressing mode to the section containing this directive.

All addresses in the section containing this directive are handled as absolute values.

This directive determines the address for storing the mnemonic code written in the line immediately after this directive. It also determines the address of the memory area to be allocated by the area allocation directive written in the line immediately after this directive.

[Examples]

.SECTIONvalue,ROMDATA .ORG0FF00H .BYTE"abcdefghijklmnopqrstuvwxyz" .ORG0FF80H .BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ" .END

The following example will generate an error because .ORG is not written immediately after .SECTION.

.SECTIONvalue,ROMDATA .BYTE"abcdefghijklmnopqrstuvwxyz" .ORG0FF80H .BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ" .END

[Remarks]

When using this directive, be sure to place it immediately after a .SECTION directive.

When .ORG is not written immediately after .SECTION, the section is handled as a relative-addressing section.

Be sure to insert a space character or a tab between this directive and the operand.

The operand should be a value from 0 to 0FFFFFFFH.

An expression or a symbol can be specified as the operand. Note that the value of the expression or symbol should be determined at assembly.

This directive must not be used in a relative-addressing section.

This directive can be used multiple times in an absolute-addressing section. Note that if the value specified as the operand is smaller than the address of the line where this directive is written, an error will be output.

This directive embeds 0 or more bytes that indicate disabling (03H) for the number of addresses preceding the location specified by the offset.

.OFFSET

This directive specifies an offset from the beginning of the section.

[Format]

OFFSET∆<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]

RENESAS

.SECTIONvalue,ROMDATA .BYTE"abcdefghijklmnopqrstuvwxyz" .OFFSET80H .BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ" .END

The following example will generate an error because the value specified in the second **.OFFSET** line is smaller than the offset to that line.

.SECTIONvalue,ROMDATA .OFFSET80H .BYTE"abcdefghijklmnopqrstuvwxyz" .OFFSET70H .BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ" .END

[Remarks]

Be sure to insert a space character or a tab between this directive and the operand.

The operand should be a value from 0 to 0FFFFFFFH.

An expression or a symbol can be specified as the operand. Note that the value of the expression or symbol should be determined at assembly.

This directive must not be used in an absolute-addressing section.

This directive can be used multiple times in a relative-addressing section. Note that if the value specified as the operand is smaller than the offset to the line where this directive is written, an error will be output.

This directive embeds 0 or more bytes that indicate disabling (03H) for the number of addresses preceding the location specified by the offset.

.ENDIAN

This directive specifies the endian for the section containing this directive. [Format]

.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]

.SECTIONvalue,ROMDATA .ORG0FF00H .ENDIANBIG .BYTE"abcdefghijklmnopqrstuvwxyz"

The following example will generate an error because .ENDIAN is not written immediately after .SECTION or .ORG.

.SECTIONvalue,ROMDATA .ORG0FF00H .BYTE"abcdefghijklmnopqrstuvwxyz" .ENDIANBIG .BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

[Remarks]

Be sure to write this directive immediately after a **.SECTION** or **.ORG** directive. Be sure to insert a space character or a tab between this directive and the operand. This directive must not be used in **CODE** sections.

.BLKB



This directive allocates a RAM area with the size specified in 1-byte units [Format]

 Δ .BLKB Δ <operand> Δ <label name:> Δ .BLKB Δ <operand>

[Description]

This directive allocates a RAM area with the size specified in 1-byte units. A label name can be defined for the address of the allocated RAM area. [Examples]

symbol.EQU 1 .SECTION area,DATA work1:.BLKB 1 work2:.BLKB symbol .BLKB symbol+1

[Remarks]

Be sure to write this directive in **DATA** sections. In section definition, write ",DATA" after a section name to specify a **DATA** section.

Be sure to insert a space character or a tab between this directive and the operand.

A numeric value, a symbol, or an expression can be specified as the operand.

The operand value should be determined at assembly.

Write a label name before this directive to define the label name for the allocated area.

Be sure to append a colon (:) to the label name.

The maximum value that can be specified for the operand is 7FFFFFFH.

.BLKW

This directive allocates 2-byte RAM areas for the specified number. [Format]

 Δ .BLKW Δ <operand> Δ <label name:> Δ .BLKW Δ <operand>

[Description]

This directive allocates 2-byte RAM areas for the specified number.

A label name can be defined for the address of the allocated RAM area. [Examples]

symbol.EQU 1 .SECTION area,DATA work1:.BLKW 1 work2:.BLKW symbol .BLKW symbol+1

[Remarks]

Be sure to write this directive in **DATA** sections. In section definition, write ",DATA" after a section name to specify a **DATA** section.

Be sure to insert a space character or a tab between this directive and the operand.

A numeric value, a symbol, or an expression can be specified as the operand.

The operand value should be determined at assembly.

Write a label name before this directive to define the label name for the allocated area.

Be sure to append a colon (:) to the label name.

The maximum value that can be specified for the operand is 3FFFFFFH.

.BLKL

This directive allocates 4-byte RAM areas for the specified number. [Format]



Δ.BLKL∆<operand> ∆<label name:>Δ.BLKL∆<operand>

[Description]

This directive allocates 4-byte RAM areas for the specified number. A label name can be defined for the address of the allocated RAM area. [Examples]

symbol.EQU 1 .SECTION area,DATA work1:.BLKL 1 work2:.BLKL symbol .BLKL symbol+1

[Remarks]

Be sure to write this directive in **DATA** sections. In section definition, write ",DATA" after a section name to specify a **DATA** section.

Be sure to insert a space character or a tab between this directive and the operand.

A numeric value, a symbol, or an expression can be specified as the operand.

The operand value should be determined at assembly.

Write a label name before this directive to define the label name for the allocated area.

Be sure to append a colon (:) to the label name.

The maximum value that can be specified for the operand is 1FFFFFFH.

.BLKD

This directive allocates 8-byte RAM areas for the specified number. [Format]

∆.BLKD∆<operand> ∆<label name:>∆.BLKD∆<operand>

[Description]

This directive allocates 8-byte RAM areas for the specified number. A label name can be defined for the address of the allocated RAM area. [Examples]

symbol.EQU 1 .SECTION area,DATA work1:.BLKD 1 work2:.BLKD symbol .BLKD symbol+1

[Remarks]

Be sure to write this directive in **DATA** sections. In section definition, write ",DATA" after a section name to specify a **DATA** section.

Be sure to insert a space character or a tab between this directive and the operand.

A numeric value, a symbol, or an expression can be specified as the operand.

The operand value should be determined at assembly.

Write a label name before this directive to define the label name for the allocated area.

Be sure to append a colon (:) to the label name.

The maximum value that can be specified for the operand is 0FFFFFFH.

.BYTE

This directive stores 1-byte fixed data in ROM. [Format]

∆.BYTE∆<operand> ∆<label name:>∆.BYTE∆<operand>



[Description]

This directive stores 1-byte fixed data in ROM. A label name can be defined for the address of the area for storing the data. [Examples] <When **endian=little** is specified>

.SECTION value,ROMDATA .BYTE 1 .BYTE "data" .BYTE symbol .BYTE symbol+1 .BYTE 1,2,3,4,5 .END

<When endian=big is specified>

.SECTION program,CODE,ALIGN=4 MOV.L R1,R2 .ALIGN 4 .BYTE 080H,00H,00H,00H .END

[Remarks]

Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.

Be sure to insert a space character or a tab between this directive and the operand.

A numeric value, a symbol, or an expression can be specified as the operand.

To specify a character or a string for the operand, enclose it within single-quotes (') or double-quotes ("). In this case, the ASCII code for the specified characters is stored.

Write a label name before this directive to define the label name for the area storing the data.

Be sure to append a colon (:) to the label name.

When the **endian=big** option is specified, this directive can be used only in the sections that satisfy the following conditions. An error will be output if this directive is used in a section that does not satisfy the conditions.

(1) ROMDATA section

.SECTION data,ROMDATA

(2) Relative-addressing CODE section for which the address alignment value is set to 4 or 8 in section definition

.SECTION program,CODE,ALIGN=4

(3) Absolute-addressing CODE section

.SECTION program,CODE .ORG 0fff00000H

To use a **.BYTE** directive in a **CODE** section while the **endian=big** option is specified, be sure to write an address correction directive (**.ALIGN 4**) in the line immediately before the **.BYTE** directive so that the data is aligned to a 4-byte boundary. If this address correction directive is not written, the assembler outputs a warning message and automatically aligns the data to a 4-byte boundary.

When the **endian=big** option is specified, the data area size in a **CODE** section must be specified to become a multiple of 4. If the data area size in a **CODE** section is not a multiple of 4, the assembler outputs a warning message and writes **NOP** (0x03) to make the data area size become a multiple of 4.

.WORD

This directive stores 2-byte fixed data in ROM. [Format]

∆.WORD∆<operand> ∆<label name:>∆.WORD∆<operand>



[Description]

This directive stores 2-byte fixed data in ROM. A label name can be defined for the address of the area for storing the data. [Examples]

.SECTION value,ROMDATA .WORD1 .WORDsymbol .WORDsymbol+1 .WORD1,2,3,4,5 .END

[Remarks]

Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.

Be sure to insert a space character or a tab between this directive and the operand.

A numeric value, a symbol, or an expression can be specified as the operand.

Neither a character nor a string can be specified for an operand.

Write a label name before this directive to define the label name for the area storing the data.

Be sure to append a colon (:) to the label name.

.LWORD

This directive stores 4-byte fixed data in ROM. [Format]

 Δ .LWORD Δ <operand> Δ <label name:> Δ .LWORD Δ <operand>

[Description]

This directive stores 4-byte fixed data in ROM. A label name can be defined for the address of the area for storing the data.

.SECTION value,ROMDATA .LWORD1 .LWORDsymbol .LWORDsymbol+1 .LWORD1,2,3,4,5 .END

[Remarks]

Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.

Be sure to insert a space character or a tab between this directive and the operand.

A numeric value, a symbol, or an expression can be specified as the operand.

Neither a character nor a string can be specified for an operand.

Write a label name before this directive to define the label name for the area storing the data.

Be sure to append a colon (:) to the label name.

.FLOAT

This directive stores 4-byte fixed data in ROM. [Format]

 Δ .FLOAT Δ <numeric value> Δ <label name:> Δ .FLOAT Δ <numeric value>

[Description]

This directive stores 4-byte fixed data in ROM.

A label name can be defined for the address of the area for storing the data. [Examples]



.FLOAT 5E2 constant: .FLOAT 5e2



[Remarks]

Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.

Specify a floating-point number as the operand.

Be sure to insert a space character or a tab between this directive and the operand.

Write a label name before this directive to define the label name for the area storing the data.

Be sure to append a colon (:) to the label name.

.DOUBLE

This directive stores 8-byte fixed data in ROM. [Format]

Δ.DOUBLE∆<numeric value> ∆<label name:>Δ.DOUBLE∆<numeric value>

[Description]

This directive stores 8-byte fixed data in ROM. A label name can be defined for the address of the area for storing the data. [Examples]

.DOUBLE 5E2 constant: .DOUBLE 5e2

[Remarks]

Be sure to use this directive in a **ROMDATA** section. To specify attribute **ROMDATA** for a section, add **,ROMDATA** after the section name when defining the section.

Specify a floating-point number as the operand.

Be sure to insert a space character or a tab between this directive and the operand.

Write a label name before this directive to define the label name for the area storing the data.

Be sure to append a colon (:) to the label name.

.ALIGN

This directive corrects the address for storing the code written in the line immediately after this directive to a multiple of two, four, or eight bytes.

[Format]

∆.ALIGN∆<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.

4.2.5 Macro Directives

These directives do not generate data corresponding to themselves but controls generation of machine code for instructions. They do not modify addresses.

These directives define macro functions and repeat macro functions.

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.

Table 4.30 Macro Directives

.MACRO

This directive defines a macro name. [Format]

 $\begin{array}{l} [macro \ definition] \\ \Delta < macro \ name > \Delta.MACRO[< parameter > [,...]] \\ \Delta body \\ \Delta.ENDM \\ [macro \ call] \\ \Delta < macro \ name > \Delta[< argument > [,...]] \end{array}$

[Description] This directive defines a macro name. It also specifies the beginning of a macro definition. [Examples: Example 1] [Macro definition example]

name.MACRO string .BYTE 'string' .ENDM

[Macro call example 1]



name"name,address"

.BYTE'name,address'

```
[Macro call example 2]
```

name(name,address)

.BYTE'(name,address)'

[Example 2]

mac .MACROp1,p2,p3 .IF ...MACPARA == 3 .IF 'p1' == 'byte' MOV.B #p2,[p3] .ELSE MOV.W #p2,[p3] .ENDIF .ELIF..MACPARA == 2 .IF 'p1' == 'byte' MOV.B #p2,[R3] .ELSE MOV.W #p2,[R3] .ENDIF .ELSE MOV.W R3.R1 .ENDIF .ENDM

macword, 10, R3; Macro call

.IF 3 == 3; Macro-expanded code .ELSE MOV.W #10,[R3] .ENDIF

[Remarks]

Be sure to specify a macro name.

For the macro name and parameter name format, refer to the Rules for Names in section 4.1.2, Names.

Use a unique name for defining each parameter, including the nested macro definitions.

To define multiple parameters, separate them by commas (,).

Make sure that all parameters specified as operands of a .MACRO directive are used in the macro body.

Be sure to insert a space character or a tab between a macro name and an argument.

Write a macro call so that the arguments correspond to the parameters on a one-to-one basis.

To use a special character in an argument, enclose it within double-quotes.

A label, a global label, and a symbol can be used in an argument.

An expression can be used in an argument.

Parameters are replaced with arguments from left to right in the order they appear.

If no argument is specified in a macro call while the corresponding parameter is defined, the assembler does not generate code for this parameter.

If there are more parameters than the arguments, the assembler does not generate code for the parameters that do not have the corresponding arguments.

When a parameter in the body is enclosed within single-quotes ('), the assembler encloses the corresponding argument within single-quotes when outputting it.

When an argument contains a comma (,) and the argument is enclosed within parentheses (()), the assembler converts the argument including the parentheses.

If there are more arguments than the parameters, the assembler does not process the arguments that do not have the corresponding parameters.

The string enclosed within double-quotes is processed as a string itself. Do not enclose parameters within doublequotes.

Up to 80 parameters can be specified within the maximum allowable number of characters for one line.

RENESAS
If the number of arguments differs from that of the parameters, the assembler outputs a warning message.

.EXITM

This directive terminates expansion of a macro body and passes control to the nearest **.ENDM**. [Format]

<macro name>Δ.MACRO Δbody Δ.EXITM Δbody Δ.ENDM

[Description]

This directive terminates expansion of a macro body and passes control to the nearest **.ENDM**. [Examples]

```
data1 .MACROvalue
.IF value == 0
.EXITM
.ELSE
.BLKBvalue
.ENDIF
.ENDM
```

data1 0 ; Macro call

```
.IF 0 == 0; Macro-expanded code
.EXITM
.ENDIF
```

[Remarks]

Write this directive in the body of a macro definition.

.LOCAL

This directive declares that the label specified as an operand is a macro local label. [Format]

.LOCAL∆<label name>[,...]

[Description]

This directive declares that the label specified as an operand is a macro local label.

Macro local labels can be specified multiple times with the same name as long as they are specified in different macro definitions or outside macro definitions.

[Examples]

name.MACRO .LOCALm1; 'm1' is macro local label m1: nop bram1 .ENDM

[Remarks]

Write this directive in a macro body.

Be sure to insert a space character or a tab between this directive and the operand.

Make sure that a macro local label is declared through this directive before the label name is defined.

For the macro local name format, refer to the Rules for Names in section 10.1.2, Names.

Multiple labels can be specified as operands of this directive by separating them by commas. Up to 100 labels can be specified in this manner.



When macro definitions are nested, a macro local label in a macro that is defined within another macro definition (outer macro) cannot use the same name as that used in the outer macro.

Up to 65,535 macro local labels can be written in one assembly source file including those used in the include files.

.ENDM

This directive specifies the end of a macro definition. [Format]

<macro name>∆.MACRO ∆body ∆.ENDM

[Description] This directive specifies the end of a macro definition. [Examples]

Ida .MACRO MOV.L #value,R3 .ENDM Ida 0 ; Expanded to MOV.L #0,R3.

.MREPEAT

This directive specifies the beginning of a repeat macro. [Format]

[<label>:]∆.MREPEAT∆<numeric value> ∆body ∆.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 .ENDR .ENDM

rep 3 ; Macro call

nop ; Macro-expanded code nop nop

[Remarks]
Be sure to specify an operand.
Be sure to insert a space character or a tab between this directive and the operand.
A label can be specified at the beginning of this directive line.
A symbol can be specified as the operand.
Forward reference symbols must not be used.
An expression can be used in the operand.
Macro definitions and macro calls can be used in the body.



The **.EXITM** directive can be used in the body.

.ENDR

This directive specifies the end of a repeat macro. [Format]

[<label>:]∆.MREPEAT∆<numeric value> ∆body ∆.ENDR

[Description]

This directive specifies the end of a repeat macro.

[Remarks] Make sure this directive corresponds to an **.MREPEAT** directive.

..MACPARA

This directive indicates the number of arguments in a macro call. [Format]

..MACPARA

[Description]

This directive indicates the number of arguments in a macro call.

This directive can be used in the body in a macro definition through .MACRO.

[Examples]

This example executes conditional assembly according to the number of macro arguments.

.GLBmem name.MACRO f1,f2 .IF..MACPARA == 2 ADD f1,f2 .ELSE ADD R3,f1 .ENDIF .ENDM

name mem ; Macro call

.ELSE ; Macro-expanded code ADD R3,mem .ENDIF

[Remarks]

This directive can be used as a term of an expression. If this directive is written outside a macro body defined through **.MACRO**, its value becomes 0.

..MACREP

This directive indicates the count of repeat macro expansions. [Format]

..MACREP

[Description]

This directive indicates the count of repeat macro expansions. This directive can be used in the body in a macro definition through **.MREPEAT**. This directive can be specified in an operand of conditional assembly. [Examples]



mac.MACRO value,reg .MREPEAT value MOV.B#0,..MACREP[reg] .ENDR .ENDM

mac3,R3; Macro call

.MREPEAT3; Macro-expanded code MOV.B#0,1[R3] MOV.B#0,2[R3] MOV.B#0,3[R3] .ENDR .ENDM

[Remarks]

This directive can be used as a term of an expression. If this directive is written outside a macro body defined through **.MACRO**, its value becomes 0.

.LEN

This directive indicates the length of the string specified as the operand. [Format]

.LEN∆{"<string>"} .LEN∆{'<string>'}

[Description]

This directive indicates the length of the string specified as the operand. [Examples]

bufset.MACRO f1 buffer:.BLKB .LEN{'f1'} .ENDM

bufset Sample ; Macro call

buffer:.BLKB 6 ; Macro-expanded code

[Remarks]

Be sure to enclose the operand within {}.

A space character or a tab can be inserted between this directive and the operand.

Characters including spaces and tabs can be specified in a string.

Be sure to enclose a string within single-quotes or double-quotes.

This directive can be used as a term of an expression.

To count the length of the macro argument, enclose the parameter name within single-quotes. When the name is enclosed within double-quotes, the length of the string specified as the parameter is counted.

.INSTR

This directive indicates the start position of a search string within a specified string.

[Format]

.INSTRA{"<string>","<search string>",<search start position> } .INSTRA{"<string>','<search string>',<search start position> }

[Description]

This directive indicates the start position of a search string within a specified string.

The position from which search is started can be specified.

[Examples]

This example detects the position (7) of string "se", counted from the beginning (top) of a specified string (japanese):



top .EQU 1 point_set.MACRO source,dest,top point.EQU .INSTR{'source','dest',top} .ENDM point set japanese,se,1 ; Macro call

point .EQU 7 ; Macro-expanded code

[Remarks]

Be sure to enclose the operand within {}.

Be sure to specify all of a string, a search string, and a search start position.

Separate the string, search string, and search start position by commas.

Neither space character nor tab can be inserted before or after a comma.

A symbol can be specified as a search start position.

When 1 is specified as the search start position, it indicates the beginning of a string.

This directive can be used as a term of an expression.

This directive is replaced with 0 when the search string is longer than the string, the search string is not found in the string, or the search start position value is larger than the length of the string.

To expand a macro by using a macro argument as the condition for detection, enclose the parameter name within single-quotes. When the name is enclosed within double-quotes, the macro is expanded by using the enclosed string as the condition for detection.

.SUBSTR

This directive extracts a specified number of characters from a specified position in a specified string. [Format]

 $SUBSTRA{"<string>",<extraction start position>,<extraction character length> }$ $SUBSTRA{ '<string>',<extraction start position>,<extraction character length> }$

[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.

4.2.6 Specific Compiler Directives

The following directives are output in some cases so that the assembler can appropriately process C language functions when the compiler generates assembly-language files.

When using the assembly-language files generated by the compiler, these directives should be used without changing the settings. These directives should not be used when creating user-created assembly-language files.

Directive	Function	
LINE_TOP	These directives are output when the functions specified by #pragma inline_asm have	
LINE_END	been expanded.	
.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.	

Table 4.31 Specific Compiler Directives

4.3 Control Instructions

This chapter describes control instructions. Control Instructions provide detailed instructions for assembler operation.

4.3.1 Outline

Control instructions provide detailed instructions for assembler operation and so are written in the source. Control instructions do not become the target of object code generation. The following table shows the types of control instructions.

Туре	Control Instructions
Assembler list directive	.LIST
Conditional assembly directives	.IF, .ELIF, .ELSE, .ENDIF
Extended function directives	.ASSERT, ?, @,FILE, .STACK, .LINE, .DEFINE

The following sections explain the details of each control instruction.

4.3.2 Assembler List Directive

This directive controls the output information and format of the assembler list file. It does not affect code generation.

.LIST

This directive can stop (**OFF**) outputting lines to the assembler list file. [Format]

$. LIST\Delta[ON|OFF]$

[Description] This directive can stop (**OFF**) outputting lines to the assembler list file.



Even in the range where line output is stopped, error lines are output to the assembler list file. This directive can start (**ON**) outputting lines to the assembler list file. When this directive is not specified, all lines are output to the assembler list file. [Examples]

.LIST ON .LIST OFF

[Remarks]

Be sure to insert a space character or a tab between this directive and the operand. Specify **OFF** as the operand to stop outputting lines. Specify **ON** as the operand to start outputting lines.

4.3.3 Conditional Assembly Directives

These directives specify whether to assemble a specified range of lines.

Table 4.32	Conditional Assembly Directives
------------	---------------------------------

Directive	Function	
.IF	Specifies the beginning of a conditional assembly block and evaluates the condition.	
.ELIF	Evaluates the second or later conditions when multiple conditional blocks are written.	
.ELSE	Specifies the beginning of a block to be assembled when all conditions are false.	
.ENDIF	Specifies the end of a conditional assembly block.	

.IF, .ELIF, .ELSE, .ENDIF

[Format]
.IF∆conditional expression body .ELIF∆conditional 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 4.33	Conditional O	perators of .IF	and .ELIF	Directives
			••••••	

Conditional Operator	Description
>	The condition is true when the lvalue is greater than the rvalue
<	The condition is true when the lvalue is smaller than the rvalue
>=	The condition is true when the lvalue is equal to or greater than the rvalue
<=	The condition is true when the lvalue is equal to or smaller than the rvalue
==	The condition is true when the lvalue is equal to the rvalue
!=	The condition is true when the lvalue is not equal to the rvalue

A conditional expression is evaluated in signed 32 bits.

Symbols can be used in the left and right sides of a conditional operator.

Expressions can be used in the left and right sides of a conditional operator. For the expression format, refer to the rules described in (2) Expression in section 4.1.5, Coding of Operands.

Strings can be used in the left and right sides of a conditional operator. Be sure to enclose a string within single-quotes (') or double-quotes (''). Strings are compared in character code values.

Examples:

"ABC" < "CBA" -> 414243 < 434241; this condition is true.

"C" < "A" -> 43 < 41; this condition is false.

Space characters and tabs can be written before and after conditional operators.

Conditional expressions can be specified in the operands of the .IF and .ELIF directives.

The assembler does not check if the evaluation result is outside the allowed range.

Forward reference symbols (reference to a symbol that is defined after this directive line) must not be specified.

If a forward reference symbol or an undefined symbol is specified, the assembler assumes the symbol value as 0 when evaluating the expression.

4.3.4 Extended Function Directives

These directives do not affect code generation.

Directive	Function
.ASSERT	Outputs a string specified in an operand to the standard error output or a file.
?	Defines and references a temporary label.
@	Concatenates strings specified before and after @ so that they are handled as one string.
FILE	Indicates the name of the assembly-language file being processed by the assembler.
.STACK	Defines a stack value for a specified symbol.
.LINE	Changes line number.
.DEFINE	Defines a replacement symbol.

.ASSERT

This directive outputs a string specified in the operand to the standard error output at assembly. [Format]



.ASSERT∆"<string>" .ASSERT∆"<string>">∆<file name> .ASSERT∆"<string>">>∆<file name>

[Description]

This directive outputs a string specified in the operand to the standard error output at assembly.

When a file name is specified, the assembler outputs the string written in the operand to the file.

When an absolute path is specified as a file name, the assembler creates a file in the specified directory.

When no absolute path is specified as a file name;

(1) if no directory information is included in the file name specified by the **output** option, the assembler creates the file specified by this directive in the current directory.

(2) if directory information is included in the file name specified by the **output** option, the assembler creates the file specified by this directive and adds the directory information for the file specified by the **output** option.

(3) if the **output** option is not specified, the assembler creates the file in the same directory containing the file specified in the command line at assembler startup.

When the **..FILE** directive is specified as a file name, the assembler creates a file in the same directory as the file specified in the command line at assembler startup.

[Examples]

To output a message to the sample.dat file:

.ASSERT "string" > sample.dat

To add a message to the sample.dat file:

.ASSERT "string" >> sample.dat

To output a message to a file with the same name as the current processing file but without a file extension:

.ASSERT "string" > ..FILE

[Remarks]

Be sure to insert a space character or a tab between the directive and the operand.

Be sure to enclose the string in the operand within double-quotes.

To output a string to a file, specify the file name after > or >>.

The symbol > directs the assembler to create a new file and output a message to the file. If a file with the same name exists, the file is overwritten.

The symbol >> directs the assembler to add the message to the contents of the specified file. If the specified file is not found, the assembler creates a new file.

Space characters or tabs can be specified before and after > and >>.

The **..FILE** directive can be specified as a file name.

?

This directive defines a temporary label. [Format]

?:

 Δ <mnemonic > Δ ?+ Δ <mnemonic > Δ ?-

[Description]

This directive defines a temporary label.

It also references the temporary label defined immediately before or after an instruction.

Definitions and references are allowed within the same file.

Up to 65,535 temporary labels can be defined in a file. In this case, if **.INCLUDE** is used in the file, the maximum number (65,535) of temporary files includes the labels in the include file.

The temporary labels converted by the assembler are output to the source list file.

[Examples]

? ◀ /BRA ?+ ▶BRA ?-**KBRA** ?-References temporary labels indicated by arrows.



[Remarks]

Write "?:" in the line that is to be defined as a temporary label.

To reference the temporary label defined immediately before an instruction, write "?-" as an operand of the instruction. To reference the temporary label defined immediately after an instruction, write "?+" as an operand of the instruction. Only the label defined immediately before or after an instruction can be referenced from the instruction.

@

This directive concatenates macro arguments, macro variables, reserved symbols, an expanded file name of directive **..FILE**, and specified strings.

[Format]

<string>@<string>[@<string> ...]

[Description]

This directive concatenates macro arguments, macro variables, reserved symbols, an expanded file name of directive **..FILE**, and specified strings.

[Examples]

Example of file name concatenation:

When the name of the currently processed file is **sample1.src**, a message is output to the **sample.dat** file in the following example.

.ASEERT "sample" > ..FILE@.dat

Example of string concatenation:

mov_nibble .MACRO p1,src,dest MOV.@p1 src,dest .ENDM

mov_nibble W,R1,R2; Macro call

MOV.W R1,R2 ;Macro-expanded code

[Remarks]

Space characters and tabs inserted before and after this directive are concatenated as a string.

Strings can be written before and after this directive.

To use @ as character data (40H), enclose it within double-quotes ("). When a string including @ is enclosed within single-quotes ('), the strings before and after @ are concatenated.

This directive can be used multiple times in one line.

To use the concatenated string as a name, do not insert space characters or tabs before or after this directive.

..FILE

This directive is expanded to the name of the file that the assembler is currently processing (assembly-language file name or include file name).

[Format]

..FILE

[Description]

This directive is expanded to the name of the file that the assembler is currently processing (assembly-language file name or include file name).

[Examples]

When the assembly-language file name is **sample.src**, a message is output to the **sample** file in the following example.

.ASSERT "sample" > ..FILE

When the assembly-language file name is sample.src, the sample.inc file is included in the following example.

.INCLUDE ...FILE@.inc



When the above line is written in the **incl.inc** file included in the **sample.src** file, a string is output to the **incl.mes** file in most cases.

.ASSERT "sample" > ..FILE@.mes

[Remarks]

This directive can be used in the operand of the **.ASSERT** and **.INCLUDE** directives. Only the file name body with neither file extension nor path is used for replacement.

.STACK

This directive defines the stack size to be used for a specified symbol referenced through the Call Walker. [Format]

.STACK∆<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 0FFFFFFCH can be specified for a stack value, and a definition with any other value is ignored.

<numeric value> must be a constant specified without using a forward reference symbol, an externally referenced symbol, or a relative address symbol.

.LINE

This directive changes the line number and file name referred to in assembler error messages or at debugging. [Format]

.LINE∆<file name>,<line number> .LINE∆<line number>

[Description]

This directive changes the line number and file name referred to in assembler error messages or at debugging. The line number and the file name specified with **.LINE** are valid until the next **.LINE** in a program.

The compiler generates **.LINE** corresponding to the line in the C source file when the assembly source program is output with the debugging option specified.

When the file name is omitted, the file name is not changed, but only the line number is changed. [Examples]

.LINE "C:\asm\test.c",5

.DEFINE

This directive defines a symbol for a string. [Format]

<symbol name>∆.DEFINE∆<string> <symbol name>∆.DEFINE∆'<string>' <symbol name>∆.DEFINE∆'<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.

4.4 Macro Names

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

Table 4.35	Predefined Macros of Compiler	Predefined Macros of	
------------	-------------------------------	----------------------	--

Option	Predefined Macro	
isa=rxv1 isa=rxv2	#defineRXV1 #defineRXV2	1 1
endian=big endian=little	#defineBIG #defineLIT	1 1
dbl_size=4 dbl_size=8	#defineDBL4 #defineDBL8	1
int_to_short	#defineINT_SHORT	1
signed_char unsigned_char	#defineSCHAR #defineUCHAR	1 1
signed_bitfield unsigned_bitfield	#defineSBIT #defineUBIT	1 1
round=zero round=nearest	#defineROZ #defineRON	1 1
denormalize=off denormalize=on	#defineDOFF #defineDON	1 1
bit_order=left bit_order=right	#defineBITLEFT #defineBITRIGHT	1 1
auto_enum	#defineAUTO_ENUM	1
library=function library=intrinsic	#defineFUNCTION_LIB #defineINTRINSIC_LIB	1
fpu	#defineFPU	1
-	#defineRENESAS (*1)	1
-	#defineRENESAS_VERSION (*1)	0xAABBCC00 (*2)
-	#defineRX (*1)	1
pic	#definePIC	1
pid	#definePID	1
cpu=rx600 cpu=rx200	#defineRX600 #defineRX200	1 1

Notes 1. Always defined regardless of the option.



Notes 2. When the version is V.AA.BB.CC, the value of **____RENESAS_VERSION__** is 0xAABBCC00. Example: For V.2.00.00, specify **#define ____RENESAS_VERSION__** 0x02000000.

Option	Predefined Macro		
isa=rxv1	RXV1	.DEFINE 1	
isa=rxv2	RXV2	.DEFINE 1	
endian=big	BIG	.DEFINE 1	
endian=little	LITTLE	.DEFINE 1	
-	RENESAS_VERSION (*1)	.DEFINE	
-	RX (*1)	.DEFINE 1	
cpu=rx600	RX600	.DEFINE 1	
cpu=rx200	RX200	.DEFINE 1	

Table 4.36 Predefined Macros of Assemble	able 4.36	Predefined Macros of Assemble
--	-----------	-------------------------------

Notes 1. Always defined regardless of the option.

Notes 2. When the version is V.AA.BB.CC, the value of **___RENESAS_VERSION__** is 0xAABBCC00. Example: For V.2.00.00, specify **___RENESAS_VERSION__** .DEFINE 02000000H.

4.5 Reserved Words

The assembler handles the same strings as assembler directives and mnemonics as reserved words. These reserved words have special functions and they cannot be used as label names or symbol names in assembly-language files. They are not case-sensitive; for example, "ABS" and "abs" are the same reserved word.

Reserved words are classified into the following types.

- 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.



4.6 Instructions

This section describes various instruction functions of RX family.

The RX CPU has short formats for frequently used instructions, facilitating the development of efficient programs that take up less memory. Moreover, some instructions are executable in one clock cycle, and this realizes high-speed arithmetic processing.

The RX CPU has a total of 90 instructions, consisting of 73 basic instructions, eight floating-point operation instructions, and nine DSP instructions.

While the RX600 Series supports all of the instructions, the RX200 Series supports the 82 instructions other than the eight for floating-point operations.

The RX CPU has 10 addressing modes, with register-register operations, register-memory operations, and bitwise operations included. Data transfer between memory locations is also possible. An internal multiplier is included for high-speed multiplication.

4.6.1 Address Space

The address space of the RX CPU is the 4 Gbyte range from address 0000 0000h to address FFFF FFFFh. Program and data regions taking up to a total of 4 Gbytes are linearly accessible. The address space of the RX-CPU is depicted in figure 1.10. For all regions, the designation may differ with the product and operating mode. For details, see the hardware manuals for the respective products.

Figure 4.1	Address Space
i iguio i.i	7 1001 000 00000



4.6.2 Register Configuration

The RX CPU has sixteen general-purpose registers, nine control registers, and one accumulator used for DSP instructions.



Figure 4.2	CPU Register Configuration
riguic 4.2	or o register configuration

	General-purp	oose registe	r	
	b31	÷		b0
		R0 (SP))*1	
		R1		
		R2		
		R3		
		R4		
		R5		
		R6		
		R7		
		R8		
		R9		
		R10		
		R11		
		R12		
		R13		
		R14		
		R15		
	b31	ISP	(Interrupt stack pointer)	ь0
			(literrup: stack pointer)	\neg
		035	(Osel stack pointer)	
		INTB	(Interrupt table register)	
		PC	(Program counter)	
		PSW	(Processor status word)	
		BPC	(Backup PC)	
		BPSW	(Backup PSW)	
		FINTV	(Fast interrupt vector register)	
		FPSW	(Floating-point status word)*2	
DSP instruction register				
DSP instruction register b63				b0

(1) General-Purpose Registers (R0 to R15)

This CPU has sixteen general-purpose registers (R0 to R15). R1 to R15 can be used as data register or address register.

R0, a general-purpose register, also functions as the stack pointer (SP). The stack pointer is switched to operate as the interrupt stack pointer (ISP) or user stack pointer (USP) by the value of the stack pointer select bit (U) in the processor status word (PSW).

(2) Interrupt Stack Pointer (ISP)/User Stack Pointer (USP) The stack pointer (SP) can be either of two types, the interrupt stack pointer (ISP) or the user stack pointer (USP). Whether the stack pointer operates as the ISP or USP depends on the value of the stack pointer select bit (U) in the processor status word (PSW). Set the ISP or USP to a multiple of four, as this reduces the numbers of cycles required to execute interrupt

Set the ISP or USP to a multiple of four, as this reduces the numbers of cycles required to execute interrupt sequences and instructions entailing stack manipulation.



- Interrupt Table Register (INTB)
 The interrupt table register (INTB) specifies the address where the relocatable vector table starts.
- (4) Program Counter (PC) The program counter (PC) indicates the address of the instruction being executed.
- (5) Backup PC (BPC) The backup PC (BPC) is provided to speed up response to interrupts. After a fast interrupt has been generated, the contents of the program counter (PC) are saved in the BPC.
- (6) Backup PSW (BPSW)

The backup PSW (BPSW) is provided to speed up response to interrupts. After a fast interrupt has been generated, the contents of the processor status word (PSW) are saved in the BPSW. The allocation of bits in the BPSW corresponds to that in the PSW.

(7) Fast Interrupt Vector Register (FINTV)

TThe fast interrupt vector register (FINTV) is provided to speed up response to interrupts. The FINTV register specifies a branch destination address when a fast interrupt has been generated.he static base register (SB) is a 16-bit register used for SB-based relative addressing.

(8) Floating-Point Status Word (FPSW)

The floating-point status word (FPSW) indicates the results of floating-point operations. In products that do not support floating-point instructions, the value "00000000h" is always read out and writing to these bits does not affect operations.

When an exception handling enable bit (Ej) enables the exception handling (Ej = 1), the corresponding Cj flag indicates the cause. If the exception handling is masked (Ej = 0), check the Fj flag at the end of a series of processing. The Fj flag is the accumulation type flag (j = X, U, Z, O, or V).

Note The FPSW is not specifiable as an operand in products of the RX200 Series.

(9) Accumulator (ACC)

The accumulator (ACC) is a 64-bit register used for DSP instructions. The accumulator is also used for the multiply and multiply-and-accumulate instructions; EMUL, EMULU, FMUL, MUL, and RMPA, in which case the prior value in the accumulator is modified by execution of the instruction.

Use the MVTACHI and MVTACLO instructions for writing to the accumulator. The MVTACHI and MVTACLO instructions write data to the higher-order 32 bits (bits 63 to 32) and the lower-order 32 bits (bits 31 to 0), respectively.

Use the MVFACHI and MVFACMI instructions for reading data from the accumulator. The MVFACHI and MVFACMI instructions read data from the higher-order 32 bits (bits 63 to 32) and the middle 32 bits (bits 47 to 16), respectively.

4.6.3 Processor Status Word (PSW)

The processor status word (PSW) indicates results of instruction execution or the state of the CPU.

4.6.4 Floating-Point Status Word (FPSW)

The floating-point status word (FPSW) indicates the results of floating-point operations. In products that do not support floating-point instructions, the value "00000000h" is always read out and writing to these bits does not affect operations.

When an exception handling enable bit (Ej) enables the exception handling (Ej = 1), the corresponding Cj flag indicates the cause. If the exception handling is masked (Ej = 0), check the Fj flag at the end of a series of processing. The Fj flag is the accumulation type flag (j = X, U, Z, O, or V).

4.6.5 Internal State after Reset is Cleared

The initial state after a reset is supervisor mode.

4.6.6 Data Types

The RX CPU can handle four types of data: integer, floating-point, bit, and string.

(1) Integer

An integer can be signed or unsigned. For signed integers, negative values are represented by two's complements.



Figure 4.3 Integer Data

Signed byte (8 bit) integer	b7 b0 S
Unsigned byte (8 bit) integer	
Signed word (16 bit) integer	b15 b0 S
Unsigned word (16 bit) integer	b15 b0
Signed long word (32 bit) integer	b31 b0 S
Unsigned long word (32 bit) integer	b31 b0
S: Sign bit	

(2) Floating-Point

Floating-point support is for the single-precision floating-point type specified in IEEE754; operands of this type can be used in eight floating-point operation instructions: FADD, FCMP, FDIV, FMUL, FSUB, FTOI, ITOF, and ROUND.

Note

Since products of the RX200 Series do not support instructions for floating-point operations, the floatingpoint exception does not occur.

The floating-point format supports the values listed below. 0 < E < 255 (normal numbers) E = 0 and F = 0 (signed zero) E = 0 and F > 0 (denormalized numbers)* E = 255 and F = 0 (infinity) E = 255 and F > 0 (NaN: Not-a-Number)

Note

* The number is treated as 0 when the DN bit in the FPSW is 1. When the DN bit is 0, an unimplemented processing exception is generated.

Figure 4.4 Floating-point Data



(3) Bitwise Operations

Five bit-manipulation instructions are provided for bitwise operations: BCLR, BMCnd, BNOT, BSET, and BTST. A bit in a register is specified as the destination register and a bit number in the range from 31 to 0. A bit in memory is specified as the destination address and a bit number from 7 to 0. The addressing modes available to specify addresses are register indirect and register relative.







(4) Strings

The string data type consists of an arbitrary number of consecutive byte (8-bit), word (16-bit), or longword (32-bit) units. Seven string manipulation instructions are provided for use with strings: SCMPU, SMOVB, SMOVF, SMOVU, SSTR, SUNTIL, and SWHILE.





4.6.7 Data Arrangement

(1) Data Arrangement in Register It shows the relation between the sizes of registers and bit numbers.



Nibble (4-bit) data	b3 b0
Byte (8-bit) data	b7 b0
Word (16-bit) data	b15 b0
Long word (32-bit) data	b31 b0

(2) Data Arrangement in Memory

Data in memory have three sizes; byte (8-bit), word (16-bit), and longword (32-bit). The data arrangement is selectable as little endian or big endian. Figure 1.7 shows the arrangement of data in memory.

Data type	Address	Data image	Data image
		(Little endian)	(Big endian)
		b7 b0	ь7 ь0
1-bit data	Address L	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Byte data	Address L	MSB LSB	MSB LSB
	6 d due		
Word data	Address M	LSB	MSB 100
	Address M+1	MSB;	
Longword data	Address N	LSB	MSB
	Address N+1		
	Address N+2		
	Address N+3	MSB	LSB

Figure 4.8 Data Arrangement in Memory

4.6.8 Vector Tables

There are two types of vector table: fixed and relocatable. Each vector in the vector table consists of four bytes and specifies the address where the corresponding exception handling routine starts.

(1) Fixed Vector Tables

The fixed vector table is allocated to a fixed address range. The individual vectors for the privileged instruction exception, access exception, undefined instruction exception, floating-point exception*, non-maskable interrupt, and reset are allocated to addresses in the range from FFFFF80h to FFFFFFFh. Figure 1.8 shows the fixed vector table.

Note

* Since products of the RX200 Series do not support instructions for floating-point operations, the floating-point exception does not occur.

	I MSB	1 1	LSB
FFFFFD0h	Privileged instruction exception		
FFFFFFD4h		Access exception	
FFFFFD8h		(Reserved)	
FFFFFDCh		Undefined instruction exception	
FFFFFE0h		(Reserved)	
FFFFFE4h	Floating-point exception		
FFFFFE8h	(Reserved)		
FFFFFECh	(Reserved)		
FFFFFF0h	(Reserved)		
FFFFFFF4h	(Reserved)		
FFFFFF8h	Non-maskable interrupt		
FFFFFFCh	Reset		

Figure 4.9 Fixed Vector Table



(2) Relocatable Vector Table

The address where the relocatable vector table is placed can be adjusted. The table is a 1,024-byte region that contains all vectors for unconditional traps and interrupts and starts at the address (IntBase) specified in the interrupt table register (INTB). Figure 1.9 shows the relocatable vector table.

Each vector in the relocatable vector table has a vector number from 0 to 255. Each of the INT instructions, which act as the sources of unconditional traps, is allocated to the vector that has the same number as that of the instruction itself (from 0 to 255). The BRK instruction is allocated to the vector with number 0. Furthermore, vector numbers within the set from 0 to 255 may also be allocated to other interrupt sources on a per-product basis.



Figure 4.10 Variable Vector Table



4.6.9 Addressing Modes

This section describes the symbols used to represent addressing modes and operations of each addressing mode.

(1) General Instruction Addressing

This addressing mode type accesses the area from address 00000h through address 0FFFh. The names of the general instruction addressing modes are as follows:

- Immediate
- Register direct
- Register indirect
- Register relative
- Post-increment register indirect
- Pre-decrement register indirect
- Indexed register indirect
- Control register direct
- PSW direct
- Program counter relative

4.6.10 Guide to This Chapter

An example illustrating how to read this chapter is shown below.



(1) Name

The name of the addressing mode.

- (2) Symbol The symbol representing the addressing mode.
- (3) Description
 A description of the addressing operation and the effective address range.
- (4) Operation diagramA diagram illustrating the addressing operation.



4.6.11 General Instruction Addressing

Immediate			
#IMM:1 #IMM:3 #IMM:4	#IMM:1 The operand is the 1-bit immediate value indicated by #IMM. This address-		ЬО
#UIMM:4 #IMM:5	ing mode is used to specify the source for the RACW instruction.	#IMM:3	b2 b0
	#IMM:3 The operand is the 3-bit immediate	#IMM:4	-2 - L0
	value indicated by #IMM. This address- ing mode is used to specify the bit number for the bit manipulation instruc-	#UIMM:4	
	tions: BCLR, BMCnd, BNOT, BSET, and BTST.	#IMM:5	
	#IMM:4 The operand is the 4-bit immediate value indicated by #IMM. This address- ing mode is used to specify the inter- rupt priority level for the MVTIPL instruction.		
	#UIMM:4 The operand is the 4-bit immediate value indicated by #UIMM after zero extension to 32 bits. This addressing mode is used to specify sources for ADD, AND, CMP, MOV, MUL, OR, and SUB instructions.		
	 #IMM:5 The operand is the 5-bit immediate value indicated by #IMM. This addressing mode is used in the following ways: to specify the bit number for the bit-manipulation instructions: BCLR, BMCnd, BNOT, BSET, and BTST; to specify the number of bit places of shifting in certain arithmetic/logic instructions: SHAR, SHLL, and SHLR; and to specify the number of bit places of rotation in certain arithmetic/logic instructions: ROTL and ROTR. 		



Immediate			
Immediate #IMM:8 #SIMM:8 #UIMM:8 #IMM:16 #SIMM:24 #IMM:32 Register Direct Rn (Rn = R0 to R15)	The specified register is the object of the operation.	When the size specifier is B #IMM:8 When the size specifier is W #SIMM:8 #UIMM:8 #SIMM:16 Sign extension b31 #SIMM:24 Sign extension #IMM:32 b31 B31 #IMM:32 B31 Rn Rn Rn Rn Register Pc	b7 b0 b8b7 b0 b8b7 b0 b8b7 b0 b8b7 b0 b0 b8b7 b0
Register Indirect			
[Rn] (Rn = R0 to R15)	The value in the specified register is the effective address of the operand. The range of valid addresses is from 00000000h to FFFFFFFh. [Rn] (Rn = R0 to R15) can be specified.	Register Rn <u>address</u>	Memory Direction of address incrementing



Register Relative		
dsp:5[Rn] (Rn = R0 to R7) dsp:8[Rn] (Rn = R0 to R15) dsp:16[Rn] (Rn = R0 to R15)	The effective address of the operand is the least significant 32 bits of the sum of the displacement (dsp) value, after zero-extension to 32 bits and multipli- cation by 1, 2, or 4 according to the specification (see the diagram at right), and the value in the specified register. The range of valid addresses is from 00000000h to FFFFFFFh. dsp:n rep- resents an n-bit long displacement value. The following mode can be specified: dsp:5[Rn] (Rn = R0 to R7), dsp:8[Rn] (Rn = R0 to R15), and dsp:16[Rn] (Rn = R0 to R7) is used only with MOV and MOVE instructions.	Register Rn address dsp \rightarrow (x) \rightarrow (+) \downarrow Direction of address incrementing · Instruction that takes a size extension specifier .B: × 1 .W: × 2 .L: × 4 · Instruction that takes a size extension specifier .B/UB: × 1 .W/UW: × 2 .L: × 4
Post-increment Re	egister Indirect	
[Rn+] (Rn = R0 to R15)	The value in the specified register is the effective address of the operand. The range of valid addresses is from 00000000h to FFFFFFFh. After the operation, 1, 2, or 4 is added to the value in the specified register accord- ing to the size specifier: .B, .W, or .L. This addressing mode is used with MOV and MOVU instructions.	Memory Register Rn address (2) (3) (3) (3) (4) (3) (4) (4) (4) (5) (4) (4) (5) (4) (5) (6) (6) (6) (7) (7) (7) (7) (7) (7) (7) (7
Pre-decrement Re	gister Indirect	
[-Rn] (Rn = R0 to R15)	According to the size specifier: .B, .W, or .L, 1, 2, or 4 is subtracted from the value in the specified register. The value after the operation is the effective address of the operand. The range of valid addresses is from 00000000h to FFFFFFFh. This addressing mode is used with MOV and MOVU instruc- tions.	When the size specifier is .B: -1 When the size specifier is .L: -4



Indexed Register	Indirect	
[Ri,Rb] (Ri = R0 to R15, Rb = R0 to R15)	The effective address of the operand is the least significant 32 bits of the sum of the value in the index register (Ri), multiplied by 1, 2, or 4 according to the size specifier: .B, .W, or .L, and the value in the base register (Rb). The range of valid addresses is from 00000000h to FFFFFFFh. This addressing mode is used with MOV and MOVU instructions.	$ \begin{array}{c} \\ Base register \\ address \\ index register \\ t \\ $
Control Register	Direct	
PC ISP USP INTB PSW BPC BPSW FINTV FPSW	The operand is the specified control register. This addressing mode is used with MVFC, MVTC, POPC, and PUSHC instructions. The PC is only selectable as the src operand of MVFC and PUSHC instructions.	Boll Register b0 PC



PSW Direct		
C Z S O I U	The operand is the specified flag or bit. This addressing mode is used with CLRPSW and SETPSW instruc- tions.	b31 b24 b23 b16 PSW b15 b8 b7 b0 PSW 0 S Z C
Program Counter	Relative	
pcdsp:3	When the branch distance specifier is .S, the effective address is the least significant 32 bits of the unsigned sum of the value in the pro- gram counter (PC) and the displace- ment (pcdsp) value. The range of the branch is from 3 to 10. The range of valid addresses is from 00000000h to FFFFFFFFh. This addressing mode is used with BCnd (where Cnd==EQ/Z or NE/NZ) and BRA instructions.	PC PC PC ++ Label pcdsp Pcdsp Hemory Branch instruction address incrementing



Program Counter	Relative	
pcdsp:8 pcdsp:16 pcdsp:24	When the branch distance specifier is .B, .W, or .A, the effective address is the signed sum of the value in the program counter (PC) and the dis- placement (pcdsp) value. The range of pcdsp depends on the branch dis- tance specifier. For .B:-128 ? pcdsp:8 ? 127 For .W:-32768 ? pcdsp:16 ? 32767 For .A:-8388608 ? pcdsp:24 ? 8388607 The range of valid addresses is from 00000000h to FFFFFFFh. When the branch distance specifier is .B, this addressing mode is used with BCnd and BRA instructions. When the branch distance specifier is .W, this addressing mode is used with BCnd (where Cnd==EQ/Z or NE/NZ), BRA, and BSR instructions. When the branch distance specifier is .A, this addressing mode is used with BRA and BSR instructions.	When the pcdsp value is negative pcdsp
Program Counter	Relative	
Rn (Rn = R0 to R15)	The effective address is the signed sum of the value in the program counter (PC) and the Rn value. The range of the Rn value is from - 2147483648 to 2147483647. The range of valid addresses is from 00000000h to FFFFFFFh. This addressing mode is used with BRA(.L) and BSR(.L) instructions.	Memory When the Rn value is negative Register Register Register When the Rn value is positive Nemory Branch instruction When the Rn value is positive



4.6.12 Instruction overview

In this chapter each instruction's syntax, operation, function, selectable src/dest, and flag changes are listed, and description examples and related instructions are shown.

An example illustrating how to read this chapter is shown below.

(1) Syntax

The syntax of the instruction using symbols. If (:format) is omitted, the assembler chooses the optimum specifier.

(a) Mnemonic**MOV**

Shows the mnemonic.

(b) Size specifier .size

Shows the data sizes in which data is handled. The following data sizes may be specified:

- .B Byte (8 bits)
- .W Word (16 bits)

.L Long word (32 bits)

Some instructions do not have a size specifier.

(c) Operands **src**, **dest**

Shows the operands.

- (d) Shows the instruction formats that can be specified in (c).
- (2) Flag change

Shows a flag change that occurs after the instruction is executed. The symbols in the table mean the following.

- "-" The flag does not change." "The flag changes depending on a condition."
- (3) Description example Description examples for the instruction.

4.6.13 Functions

It shows instructions.



ABS

Absolute value

ABS

ABSolute

```
[Syntax]]
(1)ABS dest
(2)ABS src, dest
[Operation]
(1)if ( dest < 0 )
dest = -dest;
(2)if ( src < 0 )
dest = -src;
else
dest = src;
```

[Function]

- (1)This instruction takes the absolute value of dest and places the result in dest.

- (2)This instruction takes the absolute value of src and places the result in dest.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
ABS dest	L	-	Rd	2
ABS src, dest	L	Rs	Rd	3

[Flag Change]

	Flag		С	Z	S	0
0	Charge	0	_			

Conditions

- Z : The flag is set when dest is 0 after the operation; otherwise it is cleared.
- S : The flag is set when the MSB of dest after the operation is 1; otherwise it is cleared.
- O : (1)The flag is set if dest before the operation was 80000000h; otherwise it is cleared.
 - (2)The flag is set if src before the operation was 80000000h; otherwise it is cleared.

[Description Example]

ABS R2

ABS R1, R2



ADC

Add with carry

ADC

ADd with Carry

[Syntax] ADC src, dest [Operation] dest = dest + src + C;

[Function]

- This instruction adds dest, src, and the C flag and places the result in dest.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
ADC src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	4
	L	dsp:8[Rs].L	Rd	5
	L	dsp:16[Rs].L	Rd	6

[Flag Change]

	Flag		С	Z	S	0
0	Change	0	\bigcirc			

Conditions

- C : The flag is set if an unsigned operation produces an overflow; otherwise it is cleared.
- Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.
- S : The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.
- O : The flag is set if a signed operation produces an overflow; otherwise it is cleared.

[Description Example]

ADC #127, R2 ADC R1, R2

ADC [R1], R2



ADD

Addition without carry

ADD

ADD

[Syntax] (1)ADD src, dest (2)ADD src, src2, dest

[Operation]

(1)dest = dest + src; (2)dest = src + src2;

[Function]

(1)This instruction adds dest and src and places the result in dest.(2)This instruction adds src and src2 and places the result in dest.[Instruction Format]

Syntax	Processng Size	src	src2	dest	Code size (Byte)
(1)ADD src,dst	L	#UIMM:4	-	Rd	2
	L	#SIMM:8	-	Rd	3
	L	#SIMM:16	-	Rd	4
	L	#SIMM:24	-	Rd	5
	L	#IMM:32	-	Rd	6
	L	Rs	-	Rd	2
	L	[Rs].memex	-	Rd	2 (memex == UB) 3 (memex != UB)
	L	dsp:8[Rs].m emex	-	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:16[Rs]. memex	-	Rd	4 (memex == UB) 5 (memex != UB)
(2)ADD src,src2,dst	L	#SIMM:8	Rs	Rd	3
	L	#SIMM:16	Rs	Rd	4
	L	#SIMM:24	Rs	Rd	5
	L	#IMM:32	Rs	Rd	6
	L	Rs	Rs	Rd	3

[Flag Change]

	Flag		С	Z	S	0
0	Change	0	0			

Conditions

C : The flag is set if an unsigned operation produces an overflow; otherwise it is cleared.

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S $\ \ :$ The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.

O : The flag is set if a signed operation produces an overflow; otherwise it is cleared.

[Description Example] ADD #15, R2 ADD R1, R2



ADD	[R1], R2
ADD	[R1].UB, R2
ADD	#127, R1, R2
ADD	R1, R2, R3



AND

Logical AND

AND

AND

[Syntax] (1)AND src, dest (2)AND src, src2, dest

[Operation] (1)dest = dest & src; (2)dest = src & src2;

[Function]

- (1)This instruction logically ANDs dest and src and places the result in dest.

- (2)This instruction logically ANDs src and src2 and places the result in dest.

[Instruction Format]

Syntax	Processng Size	src	src2	dest	Code size (Byte)
(1)AND src,dst	L	#UIMM:4	-	Rd	2
	L	#SIMM:8	-	Rd	3
	L	#SIMM:16	-	Rd	4
	L	#SIMM:24	-	Rd	5
	L	#IMM:32	-	Rd	6
	L	Rs	-	Rd	2
	L	[Rs].memex	-	Rd	2(memex=UB) 3(memex!=UB)
	L	dsp:8[Rs].m emex	-	Rd	3(memex==UB) 4(memex==UB)
	L	dsp:16[Rs]. memex	-	Rd	4(memex=UB) 5(memex!=UB)

Syntax	Processng Size	src	src2	dest	Code size (Byte)
(2)AND src,src2,dst	L	Rs	Rs2	Rd	3

[Flag Change]

Flag	С	Z	S	0
Change 🔿	_			_

Conditions

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S : The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.

[Description Example]

AND #15, R2 AND R1, R2



AND	[R1], R2
AND	[R1].UW, R2
AND	R1, R2, R3



BCLR

Clearing a bit

BCLR

Bit CLeaR

[Syntax] BCLR src, dest

[Operation] When dest is a memory location: unsigned char dest; dest &= ~(1 << (src & 7));

When dest is a register: register unsigned long dest; dest &= ~(1 << (src & 31)); [Function]

- This instruction clears the bit of dest, which is specified by src.

- The immediate value given as src is the number (position) of the bit.

- The range for IMM:3 operands is 0 ? IMM:3 ? 7. The range for IMM:5 is 0 ? IMM:5 ? 31.

[Instruction Format]

Syr	ntax	Processng Size	src	src2	dest	Code size (Byte)
(1)BCLR	src, dest	В	#IMM:3	-	[Rd].B	2
		В	#IMM:3	-	dsp:8[Rd].B	3
		В	#IMM:3	-	dsp:16[Rd].B	4
		В	Rs	-	[Rd].B	3
		В	Rs	-	dsp:8[Rd].B	4
		В	Rs	-	dsp:16[Rd].B	5

Syntax	Processng Size	src	src2	dest	Code size (Byte)
(2)BCLR src, dest	L	#IMM:5	-	Rd	2
	L	Rs	-	Rd	3

[Flag Change]

This instruction does not affect the states of flags.



BCnd

Relative conditional branch

BCnd

Branch Conditionally

[Syntax] BCnd(.length) src

[Operation] if (Cnd) PC = PC + src;

[Function]

- This instruction logically ANDs dest and src and stores the result in dest.

- If dest is A0 or A1 and the selected size specifier (.size) is (.B), src is zero-expanded to perform calculation in 16 bits.

BCnd		Condition	Expression
BGEU, BC	C == 1	Equal to or greater than/C flag is 1	<=
BEQ, BZ	Z == 1	Equal to/Z flag is 1	=
BGTU	(C & ~Z) == 1	Greater than	<
BPZ	S == 0	Positive or zero	> 0
BGE	(S ^ O) == 0	Equal to or greater than as signed integer	<=
BGT	((S ^ O) Z) == 0	Greater than as signed integer	<
во	O == 1	O flag is 1	
BLTU, BNC	C == 0	Less than/C flag is 0	
BNE, BNZ	Z == 0	Not equal to/Z flag is 0	
BLEU	(C & ~Z) == 0	Equal to or less than	
BN	S == 1	Negative	0>
BLE	((S ^ O) Z) == 1	Equal to or less than as signed integer	>=
BLT	(S ^ O) == 1	Less than as signed integer	>
BNO	O == 0	O flag is 0	

If src is A0 or A1, operation is performed on the eight low-order bits of A0 or A1.

[Instruction Format]

Syntax	Pro- cessng Size	src	range of pcdsp	Code size (Byte)
BEQ.S src	S	pcdsp:3	$3 \leq pcdsp \leq 10$	1
BNE.S src	S	pcdsp:3	$3 \leq pcdsp \leq 10$	1
BCnd.B src	В	pcdsp:8	-128 ≦ pcdsp ≦ 127	2


Syntax	Pro- cessng Size	SrC	range of pcdsp	Code size (Byte)
BEQ.W src	W	pcdsp:16	$-32768 \leq pcdsp \leq 32767$	3
BNE.W src	W	pcdsp:16	-32768 ≦ pcdsp ≦ 32767	3

[Flag Change]

This instruction does not affect the states of flags.

[Description Example]

BC label1

BC.B label2

Note For the RX Family assembler manufactured by Renesas Technology Corp., enter a destination address specified by a label or an effective address as the displacement value (pcdsp:3, pcdsp:8, pcdsp:16). The value of the specified address minus the address where the instruction is allocated will be stored in the pcdsp section of the instruction.

BC label

BC 1000h



BMCnd

Conditional bit transfer

BMCnd

Bit Move Conditional

[Syntax] BMCnd src, dest [Operation] (1)When dest is a memory location: unsigned char dest; if (Cnd) dest |= (1 << (src & 7)); else dest &= ~(1 << (src & 7));

(2)When dest is a register: register unsigned long dest; if (Cnd) dest |= (1 << (src & 31)); else dest &= ~(1 << (src & 31));</pre>

[Function]

This instruction moves the truth-value of the condition specified by Cnd to the bit of dest, which is specified by src; that is, 1 or 0 is transferred to the bit if the condition is true or false, respectively. The following table lists the types of BMCnd.

BCnd		Condition	Expression
BMGEU, BMC	C == 1	Equal to or greater than/C flag is 1	<=
BMEQ, BMZ	Z == 1	Equal to/Z flag is 1	=
BMGTU	(C & ~Z) == 1	Greater than	<
BMPZ	S == 0	Positive or zero	> 0
BMGE	(S ^ O) == 0	Equal to or greater than as signed integer	<=
BMGT	((S ^ O) Z) == 0	Greater than as signed integer	~
ВМО	O == 1	O flag is 1	
BMLTU, BMNC	C == 0	Less than/C flag is 0	<=
BMNE, BMNZ	Z == 0	Not equal to/Z flag is 0	
BMLEU	(C & ~Z) == 0	Equal to or less than	
BMN	S == 1	Negative	0>
BMLE	((S ^ O) Z) == 1	Equal to or less than as signed integer	>=
BMLT	(S ^ O) == 1	Less than as signed integer	>
BMNO	O == 0	O flag is 0	

- The immediate value given as src is the number (position) of the bit.



- The range for IMM:3 operands is 0 ? IMM:3 ? 7. The range for IMM:5 is 0 ? IMM:5 ? 31.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
BMCnd src, dest	В	#IMM:3	[Rd].B	3
	В	#IMM:3	dsp:8[Rd].B	4
	В	#IMM:3	dsp16[Rd].B	5

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
BMCnd src, dest	L	#IMM:5	Rd	3

[Flag Change]

This instruction does not affect the states of flags.

[Description Example]

BMC #7, [R2] BMZ #31, R2



BNOT

Inverting a bit

BNOT

Bit NOT

[Syntax] BNOT src, dest

[Operation] (1)When dest is a memory location: unsigned char dest; dest ^= (1 << (src & 7));

(2)When dest is a register: register unsigned long dest; dest ^= (1 << (src & 31));

[Function]

- This instruction inverts the value of the bit of dest, which is specified by src, and places the result into the specified bit.

- The immediate value given as src is the number (position) of the bit. The range for IMM:3 operands is $0 \le IMM:3 \le 7$. The range for IMM:5 is $0 \le IMM:5 \le 31$.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
(1)BNOT src, dest	В	#IMM:3	[Rd].B	3
	В	#IMM:3	dsp:8[Rd].B	4
	В	#IMM:3	dsp:16[Rd].B	5
	В	Rs	[Rd].B	3
	В	Rs	dsp:8[Rd].B	4
	В	Rs	dsp:16[Rd].B	5

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
(2)BNOT src, dest	L	#IMM:5	Rd	3
	L	Rs	Rd	3
	В	Rs	dsp:16[Rd].B	5

[Flag Change] This instruction does not affect the states of flags. [Description Example] BNOT #7, [R2] BNOT R1, [R2]

 BNOT
 R1, [R2]

 BNOT
 #31, R2

 BNOT
 R1, R2



BRA

Unconditional relative branch BRA

BRanch Always

[Syntax] BRA(.length) src

[Operation] PC = PC + src; [Function]

- This instruction executes a relative branch to destination address specified by src

[Instruction Format]

Syntax	Pro- cessng Size	src	Range of pcdsp/Rs	Code size (Byte)
BRA(.length) src	S	pcdsp:3	$3 \leq \text{pcdsp} \leq 10$	1
	В	pcdsp:8	-128 \leq pcdsp \leq 127	2
	W	pcdsp:16	-32768 ≦ pcdsp ≦ 32767	3
	A	pcdsp:24	-8388608 ≦ pcdsp ≦ 8388607	4
	L	Rs	-2147483648 ≦ Rs ≦ 2147483647	2

[Flag Change]

This instruction does not affect the states of flags. [Description Example]

- BRA label1

- BRA.A label2

- BRA R1
- BRA.L R2

-

Note For the RX Family assembler manufactured by Renesas Technology Corp., enter a destination address specified by a label or an effective address as the displacement value (pcdsp:3, pcdsp:8, pcdsp:16, pcdsp:24). The value of the specified address minus the address where the instruction is allocated will be stored in the pcdsp section of the instruction.

BRA label

BRA 1000h



BRK

Unconditional trap

BRK

BReaK

[Syntax] BRK

[Operation] tmp0 = PSW; U = 0; I = 0; PM = 0; tmp1 = PC + 1; PC = *IntBase; SP = SP - 4; *SP = tmp0; SP = SP - 4; *SP = tmp1;

[Function]

- This instruction generates an unconditional trap of number 0.
- This instruction causes a transition to supervisor mode and clears the PM bit in the PSW.
- This instruction clears the U and I bits in the PSW.
- The address of the instruction next to the executed BRK instruction is saved.

[Instruction Format]

Syntax	Code size (Byte)
BRK	1

[Flag Change]

This instruction does not affect the states of flags.

The state of the PSW before execution of this instruction is preserved on the stack.

[Description Example] BRK



BSET

Setting a bit

BSET

Bit SET

[Syntax] BSET src, dest

[Operation] (1)When dest is a memory location: unsigned char dest; dest |= (1 << (src & 7));

(2)When dest is a register: register unsigned long dest; dest |= (1 << (src & 31));

[Function]

- This instruction sets the bit of dest, which is specified by src.

- The immediate value given as src is the number (position) of the bit.

- The range for IMM:3 operands is 0 ? IMM:3 ? 7. The range for IMM:5 is 0 ? IMM:5 ? 31.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
(1)BSET src,dest	В	#IMM:3	[Rd].B	2
	В	#IMM:3	dsp:8[Rd].B	3
	В	#IMM:3	dsp:16[Rd].B	4
	В	Rs	[Rd].B	3
	В	Rs	dsp:8[Rd].B	4
	В	Rs	dsp:16[Rd].B	5

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
(2)BSET src,dest	L	#IMM:5	Rd	2
	L	Rs	Rd	3

[Flag Change]

This instruction does not affect the states of flags.

[Description Example]

 BSET
 #7, [R2]

 BSET
 R1, [R2]

 BSET
 #31, R2

 BSET
 R1, R2



BSR

Relative subroutine branch BSR

Branch to SubRoutine

[Syntax] BSR(.length) src

[Operation]

```
SP = SP - 4;
*SP = ( PC + n ) *;
PC = PC + src;
```

Notes 1. 1. (PC + n) is the address of the instruction following the BSR instruction.

Notes 2. 2. "n" indicates the code size. For details, refer to "Instruction Format".

[Function]

This instruction executes a relative branch to destination address specified by src.

[Instruction Format]

Syntax	Pro- cessng Size	src	Range of pcdsp / Rs	Code size (Byte)
BSR(.length) src	W	pcdsp:16	-32768 ≦ pcdsp ≦ 32767	3
	A	pcdsp:24	-8388608 ≦ pcdsp ≦ 8388607	4
	L	Rs	-2147483648 ≦ Rs ≦ 2147483647	2

[Flag Change] This instruction does not affect the states of flags. [Description Example] BSR label1 BSR.A label2 BSR R1 BSR.L R2

Note For the RX Family assembler manufactured by Renesas Technology Corp., enter a destination address specified by a label or an effective address as the displacement value (pcdsp:16, pcdsp:24). The value of the specified address minus the address where the instruction is allocated will be stored in the pcdsp section of the instruction.

BSR label

BSR 1000h



BTST

Testing a bit

BTST

Bit TeST

[Syntax] BTST src, src2

[Operation] (1)When src2 is a memory location: unsigned char src2; Z = ~((src2 >> (src & 7)) & 1); C = ((src2 >> (src & 7)) & 1);

(2)When src2 is a register: register unsigned long src2; Z = ~((src2 >> (src & 31)) & 1); C = ((src2 >> (src & 31)) & 1);

[Function]

- This instruction moves the inverse of the value of the bit of scr2, which is specified by src, to the Z flag and the value of the bit of scr2, which is specified by src, to the C flag.
- The immediate value given as src is the number (position) of the bit.
- The range for IMM:3 operands is 0 ? IMM:3 ? 7. The range for IMM:5 is 0 ? IMM:5 ? 31.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	Code size (Byte)
(1)BTST src, src2	В	#IMM:3	[Rs2].B	2
	В	#IMM:3	dsp:8[Rs2].B	3
	В	#IMM:3	dsp:16[Rs2].B	4
	В	Rs	[Rs2].B	3
	В	Rs	dsp:8[Rs2].B	4
	В	Rs	dsp16:[Rs2].B	5

Syntax	Pro- cessng Size	src	src2	Code size (Byte)
(2)BTST src, src2	L	#IMM:5	Rs2	2
	L	Rs	Rs2	3

[Flag Change]

Flag	С	Z	S	0
Change	_	_	_	_

Conditions

C : The flag is set if the specified bit is 1; otherwise it is cleared.

Z : The flag is set if the specified bit is 0; otherwise it is cleared.

[Description Example] BTST #7, [R2] BTST R1, [R2] BTST #31, R2 BTST R1, R2



CLRPSW

Clear a flag or bit in the PSW CLRPSW CLeaR flag in PSW

[Syntax] CLRPSW dest

[Operation] dest = 0;

[Function]

- This instruction clears the O, S, Z, or C flag, which is specified by dest, or the U or I bit.
- In user mode, writing to the U or I bit is ignored. In supervisor mode, all flags and bits can be written to.

[Instruction Format]

Syntax	dest	Code size (Byte)
CLRPSW dest	flag	2

[Flag Change]

Flag	С	Z	S	0
Change	*	*	*	*

Note *: The specified flag becomes 0.

[Description Example] CLRPSW C CLRPSW Z



CMP

Comparison

CMP

CoMPare

[Syntax] СМР src, src2

[Operation] src2 - src;

[Function]

- This instruction changes the states of flags in the PSW to reflect the result of subtracting src from src2.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	Code size (Byte)
CMP src, src2	L	#UIMM:4	Rs	2
	L	#UIMM:8(注1)	Rs	3
	L	#SIMM:8(注 1)	Rs	3
	L	#SIMM:16	Rs	4
	L	#SIMM:24	Rs	5
	L	#IMM:32	Rs	6
	L	Rs	Rs2	2
	L	[Rs].memex	Rs2	2(memex == UB) 3(memex != UB)
	L	dsp:8[Rs].memex(注 2)	Rs2	3(memex == UB) 4(memex != UB)
	L	dsp:16[Rs].memex(注 2)	Rs2	4(memex == UB) 5(memex == UB)

[Flag Change]

Flag		С	Z	S	0	
0	Charge	0	0			

Conditions

C : The flag is set if an unsigned operation does not produce an overflow; otherwise it is cleared.

Ζ

: The flag is set if the result of the operation is 0; otherwise it is cleared.: The flag is set if the MSB of the result of the operation is 1; otherwise it is cleared. S

O : The flag is set if a signed operation produces an overflow; otherwise it is cleared.

[Description Example]

CMP #7, R2 CMP R1, R2 CMP [R1], R2



DIV

Signed division

DIV

DIVide

[Syntax] DIV src, dest

[Operation] dest = dest / src;

[Function]

- This instruction divides dest by src as signed values and places the quotient in dest. The quotient is rounded towards 0.

- The calculation is performed in 32 bits and the result is placed in 32 bits.

- The value of dest is undefined when the divisor (src) is 0 or when overflow is generated after the operation.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
DIV src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].memex	Rd	3(memex == UB) 4(memex != UB)
	L	dsp:8[Rs].memex(注)	Rd	4(memex == UB) 5(memex != UB)
	L	dsp:16[Rs].memex(注)	Rd	5(memex == UB) 6(memex != UB)

[Flag Change]

Flag	С	Z	S	0
⊖ Change	_	_	_	

Conditions

O : This flag is set if the divisor (src) is 0 or the calculation is -2147483648 / -1; otherwise it is cleared.

[Description Example]

DIV #10, R2

DIV R1, R2

DIV [R1], R2

DIV 3[R1].B, R2



DIVU

Unsigned division DIVide Unsigned DIVU

[Syntax] DIVU src, dest

[Operation] dest = dest / src;

[Function]

- This instruction divides dest by src as unsigned values and places the quotient in dest. The quotient is rounded towards 0.
- The calculation is performed in 32 bits and the result is placed in 32 bits.

- The value of dest is undefined when the divisor (src) is 0.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
DIVU src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].memex	Rd	3(memex == UB) 4(memex != UB)
	L	dsp:8[Rs].memex(注)	Rd	4(memex == UB) 5(memex != UB)
	L	dsp:16[Rs].memex(注)	Rd	5(memex == UB) 6(memex != UB)

[Flag Change]

Flag	С	Z	F	0
Change	_	_	_	

[Condition]

O:The flag is set if the divisor (src) is 0; otherwise it is cleared.

[Description Example]

DIVU #10, R2 DIVU R1, R2

DIVU [R1], R2

DIVU 3[R1].UB, R2



EMUL

Signed multiplication

EMUL

Extended MULtiply, signed

[Syntax] EMUL src, dest

[Operation] dest2:dest = dest * src;

[Function]

- This instruction multiplies dest by src, treating both as signed values.

- The calculation is performed on src and dest as 32-bit operands to obtain a 64-bit result, which is placed in the register pair, dest2:dest (R(n+1):Rn).

- Any of the 15 general registers (Rn (n: 0 to 14)) is specifiable for dest.

Note The accumulator (ACC) is used to perform the function. The value of ACC after executing the instruction is undefined.

Register Specified for dest	Registers Used for 64-Bit Extension
R0	R1:R0
R1	R2:R1
R2	R3:R2
R3	R4:R3
R4	R5:R4
R5	R6:R5
R6	R7:R6
R7	R8:R7
R8	R9:R8
R9	R10:R9
R10	R11:R10
R11	R12:R11
R12	R13:R12
R13	R14:R13
R14	R15:R14



[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
EMUL src, dest	L	#SIMM:8	Rd (Rd=R0~R14)	4
	L	#SIMM:16	Rd (Rd=R0~R14)	5
	L	#SIMM:24	Rd (Rd=R0~R14)	6
	L	#IMM:32	Rd (Rd=R0~R14)	7
	L	Rs	Rd (Rd=R0~R14)	3
	L	[Rs].memex	Rd (Rd=R0~R14)	3(memex == UB) 4(memex != UB)
	L	dsp:8[Rs].memex(注)	Rd (Rd=R0~R14)	4(memex == UB) 5(memex != UB)
	L	dsp:16[Rs]memex(注)	Rd (Rd=R0~R14)	5(memex == UB) 6(memex != UB)

[Flag Change]

This instruction does not affect the states of flags.

[Description Example]

EMUL #10, R2 EMUL R1, R2 EMUL [R1], R2 EMUL 8[R1].W, R2



EMULU

Unsigned multiplication EMULU Extended MULtiply, Unsigned

[Syntax] EMULU src, dest

[Operation] dest2:dest = dest * src;

[Function]

- This instruction multiplies dest by src, treating both as unsigned values.
- The calculation is performed on src and dest as 32-bit operands to obtain a 64-bit result, which is placed in the register pair, dest2:dest (R(n+1):Rn).
- Any of the 15 general registers (Rn (n: 0 to 14)) is specifiable for dest.
- Note The accumulator (ACC) is used to perform the function. The value of ACC after executing the instruction is undefined.

Register Specified for dest	Registers Used for 64-Bit Extension
R0	R1:R0
R1	R2:R1
R2	R3:R2
R3	R4:R3
R4	R5:R4
R5	R6:R5
R6	R7:R6
R7	R8:R7
R8	R9:R8
R9	R10:R9
R10	R11:R10
R11	R12:R11
R12	R13:R12
R13	R14:R13
R14	R15:R14

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
EMUL U src, dest	L	#SIMM:8	Rd (Rd=R0~R14)	4
	L	#SIMM:16	Rd (Rd=R0~R14)	5
	L	#SIMM:24	Rd (Rd=R0~R14)	6
	L	#IMM:32	Rd (Rd=R0~R14)	7



Syntax	Pro- cessng Size	src	dest	Code size (Byte)
	L	Rs	Rd (Rd=R0~R14)	3
	L	[Rs].memex	Rd (Rd=R0~R14)	3(memex == UB) 4(memex != UB)
	L	dsp:8[Rs].memex(注)	Rd (Rd=R0~R14)	4(memex == UB) 5(memex != UB)
	L	dsp:16[Rs]memex(注)	Rd (Rd=R0~R14)	5(memex == UB) 6(memex != UB)

[Flag Change]

This instruction does not affect the states of flags.

[Description Example]

EMULU #10, R2 EMULU R1, R2 EMULU [R1], R2 EMULU 8[R1].UW, R2



FADD

Floating-point addition Floating-point ADD

FADD

[Syntax] FADD src, dest

[Operation] FADD src, dest

[Function]

- This instruction adds the single-precision floating-point numbers stored in dest and src and places the result in dest. Rounding of the result is in accord with the setting of the RM[1:0] bits in the FPSW.
- Handling of denormalized numbers depends on the setting of the DN bit in the FPSW.
- The operation result is +0 when the sum of src and dest of the opposite signs is exactly 0 except in the case of a rounding mode towards -?. The operation result is -0 when the rounding mode is towards -?.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
FADD src, dest	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L(注)	Rd	4
	L	dsp:16[Rs].L(注)	Rd	5

[Flag Change]

Flag	С	Ø	S	0	CV	CZ	CU	СХ	CE	FV	FO	FZ	FU	FX
Change	Ι			Ι	0	0	0	0	0	\bigcirc	\bigcirc	_	\bigcirc	\bigcirc

Conditions

- Z : The flag is set if the result of the operation is +0 or -0; otherwise it is cleared.
- S : The flag is set if the sign bit (bit 31) of the result of the operation is 1; otherwise it is cleared.
- CV :: The flag is set if an invalid operation exception is generated; otherwise it is cleared.
- CO : : The flag is set if an overflow exception is generated; otherwise it is cleared.
- CZ :: The value of the flag is always 0.
- CU :: The flag is set if an underflow exception is generated; otherwise it is cleared.
- CX :: The flag is set if an inexact exception is generated; otherwise it is cleared.
- CE :: The flag is set if an unimplemented processing is generated; otherwise it is cleared.
- FV :: The flag is set if an invalid operation exception is generated, and otherwise left unchanged.
- FO :: The flag is set if an overflow exception is generated, and otherwise left unchanged.
- FU :: The flag is set if an underflow exception is generated, and otherwise left unchanged.
- FX :: The flag is set if an inexact exception is generated, and otherwise left unchanged.
- Note The FX, FU, FO, and FV flags do not change if any of the exception enable bits EX, EU, EO, and EV is 1. The S and Z flags do not change when an exception is generated.

[Description Example]

FADD R1, R2

FADD [R1], R2



FCMP

Floating-point comparison

FCMP

Floating-point CoMPare

[Syntax] src2 - src;

[Operation] src2 - src;

[Function]

- This instruction compares the single-precision floating numbers stored in src2 and src and changes the states of flags according to the result.
- Handling of denormalized numbers depends on the setting of the DN bit in the FPSW.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	Code size (Byte)
FCMP src, src2	L	#IMM:32	Rs2	7
	L	Rs	Rs2	3
	L	[Rs].L	Rs2	3
	L	dsp:8[Rs].L(注)	Rs2	4
	L	dsp:16[Rs].L(注)	Rs2	5

[Flag Change]

Flag	С	Z	S	0	CV	CZ	CU	СХ	CE	FV	FO	FZ	FU	FX
Change	-		0	0	0	0	0	0	0	Ð	-	_	_	_

Conditions

Z : The flag is set if src2 == src; otherwise it is cleared.

S : The flag is set if src2 < src; otherwise it is cleared.
 O :The flag is set if an ordered classification based on the comparison result is impossible; otherwise it is cleared.

CV :: The flag is set if an invalid operation exception is generated; otherwise it is cleared.

CO : : The value of the flag is always 0.

CZ :: The value of the flag is always 0.

CU :: The value of the flag is always 0.

CX :: The value of the flag is always 0.

CE :: The flag is set if an unimplemented processing exception is generated; otherwise it is cleared.

FV :: The flag is set if an invalid operation exception is generated; otherwise it does not change.

The FV flag does not change if the exception enable bit EV is 1. The O, S, and Z flags do not change Note when an exception is generated.

[Description Example]

FCMP R1, R2 FCMP [R1], R2



FDIV

Floating-point division Floating-point DIVide FDIV

[Syntax] FDIV src, dest

[Operation] dest = dest / src;

[Function]

- This instruction divides the single-precision floating-point number stored in dest by that stored in src and places the result in dest. Rounding of the result is in accord with the setting of the RM[1:0] bits in the FPSW.
- Handling of denormalized numbers depends on the setting of the DN bit in the FPSW.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
FDIV src, dest	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L(注)	Rd	4
	L	dsp:16[Rs].L(注)	Rd	5

[Flag Change]

	Flag	С	Z	S	0	CV	CZ	CU	СХ	CE	FV	FO	FZ	FU	FX
0	Change 🔾	ð	0	0	Ą	0	0	0		0	0				

Conditions

- Z : The flag is set if the result of the operation is +0 or -0; otherwise it is cleared.
- S :The flag is set if the sign bit (bit 31) of the result of the operation is 1; otherwise it is cleared.
- CV :: The flag is set if an invalid operation exception is generated; otherwise it is cleared.
- CO :: The flag is set if an overflow exception is generated; otherwise it is cleared.
- CZ :: The flag is set if a division-by-zero exception is generated; otherwise it is cleared.
- CU :: The flag is set if an underflow exception is generated; otherwise it is cleared.

CX :: The flag is set if an inexact exception is generated; otherwise it is cleared.

- CE :: The flag is set if an unimplemented processing exception is generated; otherwise it is cleared.
- FV :: The flag is set if an invalid operation exception is generated; otherwise it does not change.

FO :: The flag is set if an overflow exception is generated; otherwise it does not change.

- FZ :: The flag is set if a division-by-zero exception is generated; otherwise it does not change.
- FU :: The flag is set if an underflow exception is generated; otherwise it does not change.
- FX :: The flag is set if an inexact exception is generated; otherwise it does not change.
- Note The FX, FU, FZ, FO, and FV flags do not change if any of the exception enable bits EX, EU, EZ, EO, and EV is 1. The S and Z flags do not change when an exception is generated.

[Description Example] FDIV R1, R2 FDIV [R1], R2



FMUL

Floating-point multiplication FMUL Floating-point MULtiply

[Syntax] FMUL src, dest

[Operation] dest = dest * src;

[Function]

- This instruction multiplies the single-precision floating-point number stored in dest by that stored in src and places the result in dest. Rounding of the result is in accord with the setting of the RM[1:0] bits in the FPSW.
- Handling of denormalized numbers depends on the setting of the DN bit in the FPSW.
- The accumulator (ACC) is used to perform the function. The value of ACC after executing the instruction is undefined.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
FMUL src, dest	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L(注)	Rd	4
	L	dsp:16[Rs].L(注)	Rd	5

[Flag Change]

	Flag	С	Z	S	0	CV	CZ	CU	СХ	CE	FV	FO	FZ	FU	FX
\bigcirc	Change 🔾	Ð	0	0	Ð	0	0	0		0	0				

Conditions

- Z : The flag is set if the result of the operation is +0 or -0; otherwise it is cleared.
- S : The flag is set if the sign bit (bit 31) of the result of the operation is 1; otherwise it is cleared.
- CV : The flag is set if an invalid operation exception is generated; otherwise it is cleared.
- CO : The flag is set if an overflow exception is generated; otherwise it is cleared.
- CZ : The flag is set if a division-by-zero exception is generated; otherwise it is cleared.
- CU : The flag is set if an underflow exception is generated; otherwise it is cleared.

CX :The flag is set if an inexact exception is generated; otherwise it is cleared.

CE :The flag is set if an unimplemented processing exception is generated; otherwise it is cleared.

FV :The flag is set if an invalid operation exception is generated; otherwise it does not change.

FO :The flag is set if an overflow exception is generated; otherwise it does not change.

FZ :The flag is set if a division-by-zero exception is generated; otherwise it does not change.

FU :The flag is set if an underflow exception is generated; otherwise it does not change.

FX : The flag is set if an inexact exception is generated; otherwise it does not change.

Note The FX, FU, FZ, FO, and FV flags do not change if any of the exception enable bits EX, EU, EZ, EO, and EV is 1. The S and Z flags do not change when an exception is generated.

[Description Example]

FMUL R1, R2

FMUL [R1], R2



FSUB

Floating-point subtraction Floating-point SUBtract

FSUB

[Syntax] FSUB src, dest

[Operation] dest = dest - src;

[Function]

- This instruction subtracts the single-precision floating-point number stored in src from that stored in dest and places the result in dest. Rounding of the result is in accord with the setting of the RM[1:0] bits in the FPSW.
- Handling of denormalized numbers depends on the setting of the DN bit in the FPSW.
- The operation result is +0 when subtracting src from dest with both the same signs is exactly 0 except in the case of a rounding mode towards -?. The operation result is -0 when the rounding mode is towards -?.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
FSUB src, dest	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L(注)	Rd	4
	L	dsp:16[Rs].L(注)	Rd	5

[Flag Change]

Flag	С	Z	S	0	CV	CZ	CU	СХ	CE	FV	FO	FZ	FU	FX
Change	Ģ	0	0	Ą	0	0	0		0	0		_		

Conditions

- Z : The flag is set if the result of the operation is +0 or -0; otherwise it is cleared.
- S : The flag is set if the sign bit (bit 31) of the result of the operation is 1; otherwise it is cleared.
- CV : The flag is set if an invalid operation exception is generated; otherwise it is cleared.
- CO :The flag is set if an overflow exception is generated; otherwise it is cleared.
- CZ : The value of the flag is always 0.
- CU :The flag is set if an underflow exception is generated; otherwise it is cleared.
- CX :The flag is set if an inexact exception is generated; otherwise it is cleared.
- CE :The flag is set if an unimplemented processing exception is generated; otherwise it is cleared.
- FV :The flag is set if an invalid operation exception is generated; otherwise it does not change.
- FO :The flag is set if an overflow exception is generated; otherwise it does not change.
- FU :The flag is set if an underflow exception is generated; otherwise it does not change.
- $\mathsf{FX}\;$: The flag is set if an inexact exception is generated; otherwise it does not change.

Note The FX, FU, FO, and FV flags do not change if any of the exception enable bits EX, EU, EO, and EV is 1. The S and Z flags do not change when an exception is generated.

[Description Example]

FSUB R1, R2

FSUB [R1], R2



FTOI

Floating point to integer conver- FTOI sion

Float TO Integer

[Syntax] FTOI src, dest

[Operation] dest = (signed long) src;

[Function]

- This instruction converts the single-precision floating-point number stored in src into a signed longword (32-bit) integer and places the result in dest.

- The result is always rounded towards 0, regardless of the setting of the RM[1:0] bits in the FPSW.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
FTOI src, dest	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L(注)	Rd	4
	L	dsp:16[Rs].L(注)	Rd	5

[Flag Change]

Flag	С	Z	S	0	CV	CZ	CU	СХ	CE	FV	FO	FZ	FU	FX
O Change	-	0	0	$-\!$	0	0	0		0	0	-	-	-	

Conditions

Z : The flag is set if the result of the operation is 0; otherwise it is cleared.

S : The flag is set if the sign bit (bit 31) of the result of the operation is 1; otherwise it is cleared.

CV : The flag is set if an invalid operation exception is generated; otherwise it is cleared.

CO : The value of the flag is always 0.

CZ : The value of the flag is always 0.

CU : The value of the flag is always 0.

CX :The flag is set if an inexact exception is generated; otherwise it is cleared.

CE :The flag is set if an unimplemented processing exception is generated; otherwise it is cleared.

FV :The flag is set if an invalid operation exception is generated; otherwise it does not change.

FX : The flag is set if an inexact exception is generated; otherwise it does not change.

Note The FX and FV flags do not change if any of the exception enable bits EX and EV is 1. The S and Z flags do not change when an exception is generated.

[Description Example]

FTOI R1, R2

FTOI [R1], R2



INT

Software interrupt

INT

INTerrupt

[Syntax] INT src [Operation] tmp0 = PSW; U = 0; I = 0; PM = 0; tmp1 = PC + 3; PC = *(IntBase + src * 4); SP = SP - 4; *SP = tmp0; SP = SP - 4; *SP = tmp1;

[Function]

- This instruction generates the unconditional trap which corresponds to the number specified as src.
- The INT instruction number (src) is in the range 0 ? src ? 255.
- This instruction causes a transition to supervisor mode, and clears the PM bit in the PSW to 0.
- This instruction clears the U and I bits in the PSW to 0.

[Instruction Format]

	Syntax	SrC	Code size (Byte)
INT	src	#IMM:8	3

[Flag Change]

- This instruction does not affect the states of flags.

- The state of the PSW before execution of this instruction is preserved on the stack.

[Description Example]

INT #0



ITOF

Integer to floating-point conver- ITOF sion

Integer TO Floating-point

[Syntax] ITOF src, dest

[Operation] dest = (float) src;

[Function]

- This instruction converts the signed longword (32-bit) integer stored in src into a single-precision floating-point number and places the result in dest. Rounding of the result is in accord with the setting of the RM[1:0] bits in the FPSW. 00000000h is handled as +0 regardless of the rounding mode.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
ITOF src, dest	L	Rs	Rd	3
	L	[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:8[Rs].memex(注)	Rd	4 (memex == UB) 5 (memex != UB)
	L	dsp:16[Rs].memex(注)	Rd	5 (memex == UB) 6 (memex != UB)

[Flag Change]

Flag	С	Z	S	0	CV	CZ	CU	СХ	CE	FV	FO	FZ	FU	FX
O Change	-		0	-0	0	0	0		0	-0	-	_	-	

Conditions

Z :The flag is set if the result of the operation is +0; otherwise it is cleared.

S :The flag is set if the sign bit (bit 31) of the result of the operation is 1; otherwise it is cleared.

CV : The value of the flag is always 0.

CO :The value of the flag is always 0.CZ:The value of the flag is always 0.

CU : The value of the flag is always 0.

CX :The flag is set if an inexact exception is generated; otherwise it is cleared.

CE:The value of the flag is always 0.

FX:The flag is set if an inexact exception is generated; otherwise it does not change.

Note The FX flag does not change if the exception enable bit EX is 1. The S and Z flags do not change when an exception is generated.

[Description Example]

ITOF R1, R2

ITOF [R1], R2

ITOF 16[R1].L, R2



JMP

Unconditional jump

JMP

JuMP

[Syntax] JMP src

[Operation] PC = src;

[Function] This instruction branches to the instruction specified by src.

[Instruction Format]

	Syntax	src	Code size (Byte)
JMP	src	Rs	2

[Flag Change] This instruction does not affect the states of flags.

[Description Example] JMP R1



JSR

Jump to a subroutine JSR

Jump SubRoutine

[Syntax] JSR src

[Operation] SP = SP - 4; *SP = (PC + 2);* PC = src;

[Function]

- This instruction causes the flow of execution to branch to the subroutine specified by src.

[Instruction Format]

Syntax	src	Code size (Byte)
JSR src	Rs	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] JSR R1



MACHI

Multiply-Accumulate the high- MACHI order word

Multiply-ACcumulate HIgh-order word

[Syntax] MACHI src, src2

[Operation] signed short tmp1, tmp2; signed long long tmp3; tmp1 = (signed short) (src >> 16); tmp2 = (signed short) (src2 >> 16); tmp3 = (signed long) tmp1 * (signed long) tmp2; ACC = ACC + (tmp3 << 16);

[Function]

- This instruction multiplies the higher-order 16 bits of src by the higher-order 16 bits of src2, and adds the result to the value in the accumulator (ACC). The addition is performed with the least significant bit of the result of multiplication corresponding to bit 16 of ACC. The result of addition is stored in ACC. The higher-order 16 bits of src and the higher-order 16 bits of src2 are treated as signed integers.

[Instruction Format]

Syntax	src	src2	Code size (Byte)
MACHI src, src2	Rs	Rs2	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MACHI R1, R2



MACLO

Multiply-Accumulate the low- MACLO order word

Multiply-ACcumulate LOw-order word

[Syntax] MACLO src, src2

[Operation] signed short tmp1, tmp2; signed long long tmp3; tmp1 = (signed short) src; tmp2 = (signed short) src2; tmp3 = (signed long) tmp1 * (signed long) tmp2; ACC = ACC + (tmp3 << 16);

[Function]

- This instruction multiplies the lower-order 16 bits of src by the lower-order 16 bits of src2, and adds the result to the value in the accumulator (ACC). The addition is performed with the least significant bit of the result of multiplication corresponding to bit 16 of ACC. The result of addition is stored in ACC. The lower-order 16 bits of src and the lower-order 16 bits of src2 are treated as signed integers.

[Instruction Format]

Syntax	src	src2	Code size (Byte)
MACLO src, src2	Rs	Rs2	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MACLO R1, R2



MAX

Selecting the highest value MAX

MAXimum value select

[Syntax] MAX src, dest

[Operation] if (src > dest) dest = src;

[Function]

- This instruction compares src and dest as signed values and places whichever is greater in dest.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
MAX src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:8[Rs].memex(注)	Rd	4 (memex == UB) 5 (memex != UB)
	L	dsp:16[Rs].memex(注)	Rd	5 (memex == UB) 6 (memex != UB)

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example]

 MAX
 #10, R2

 MAX
 R1, R2

 MAX
 [R1], R2

 MAX
 3[R1].B, R2



MIN

Selecting the lowest value

MIN

MINimum value select

[Syntax] MIN src, dest

[Operation] if (src < dest) dest = src;

[Function]

- This instruction compares src and dest as signed values and places whichever is smaller in dest.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
MIX src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:8[Rs].memex(注)	Rd	4 (memex == UB) 5 (memex != UB)
	L	dsp:16[Rs].memex(注)	Rd	5 (memex == UB) 6 (memex != UB)

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example]

MIN #10, R2 MIN R1, R2

MIN [R1], R2 MIN 3[R1].B, R2



MOV

Transferring data

MOV

MOVe

[Syntax] MOV.size src, dest

[Operation] dest = src;

[Function]

- This instruction transfers src to dest as listed in the following table.

[Instruction Format]

Syntax	size	Processng Size	src	dest	Code size (Byte)
MOV.size src, dest	B/W/L	size	Rs (Rs=R0 ~ R7)	dsp:5[Rd](注 1) (Rd=R0 ~ R7)	2
	B/W/L	L	dsp:5[Rs](注 1) (Rs=R0 ~ R7)	Rd (Rd=R0 ~ R7)	2
	L	L	#UIMM:4	Rd	2
	В	В	#IMM:8	dsp:5[Rd] (注 1) (Rd=R0 ~ R7)	3
	W/L	size	#UIMM:8	dsp:5[Rd] (注 1) (Rd=R0 ~ R7)	3
	L	L	#UIMM:8(注 2)	Rd	3
	L	L	#SIMM:8(注 2)	Rd	3
	L	L	#SIMM:16	Rd	4
	L	L	#SIMM:24	Rd	5
	L	L	#IMM:32	Rd	6
	B/W	L	Rs	Rd	2
	L	L	Rs	Rd	3
	В	В	#IMM:8	[Rd]	3
	В	В	#IMM:8	dsp:8[Rd](注 1)	4
	В	В	#IMM:8	dsp:16[Rd](注 1)	5
	W	W	#SIMM:8	[Rd]	3
	W	W	#SIMM:8	dsp:8[Rd](注 1)	4
	W	W	#SIMM:8	dsp:16[Rd](注 1)	5
	W	W	#IMM:16	[Rd]	4
	W	W	#IMM:16	dsp:8[Rd](注 1)	5
	W	W	#IMM:16	dsp:16[Rd](注 1)	6
	L	L	#SIMM:8	[Rd]	3
	L	L	#SIMM:8	dsp:8[Rd](注 1)	4



Syntax	size	Processng Size	src	dest	Code size (Byte)
	L	L	#SIMM:8	dsp:16[Rd](注 1)	5
	L	L	#SIMM:16	[Rd]	4
	L	L	#SIMM:16	dsp:8[Rd](注 1)	5
	L	L	#SIMM:16	dsp:16[Rd](注 1)	6
	L	L	#SIMM:24	[Rd]	5
	L	L	#SIMM:24	dsp:8[Rd](注 1)	6
	L	L	#SIMM:24	dsp:16[Rd](注 1)	7
	L	L	#IMM:32	[Rd]	6
	L	L	#IMM:32	dsp:8[Rd](注 1)	7
	L	L	#IMM:32	dsp:16[Rd](注 1)	8
	B/W/L	L	[Rs]	Rd	2
	B/W/L	L	dsp:8[Rs](注 1)	Rd	3
	B/W/L	L	dsp:16[Rs](注 1)	Rd	4
	B/W/L	L	[Ri, Rb]	Rd	3
	B/W/L	size	Rs	[Rd]	2
	B/W/L	size	Rs	dsp:8[Rd](注 1)	3
	B/W/L	size	Rs	dsp:16[Rd](注 1)	4
	B/W/L	size	Rs	[Ri, Rb]	3
	B/W/L	size	[Rs]	[Rd]	2
	B/W/L	size	[Rs]	dsp:8[Rd](注 1)	3
	B/W/L	size	[Rs]	dsp:16[Rd](注 1)	4
	B/W/L	size	dsp:8[Rs](注 1)	[Rd]	3
	B/W/L	size	dsp:8[Rs](注 1)	dsp:8[Rd](注 1)	4
	B/W/L	size	dsp:8[Rs](注 1)	dsp:16[Rd](注 1)	5
	B/W/L	size	dsp:16[Rs](注 1)	[Rd]	4
	B/W/L	size	dsp:16[Rs](注 1)	dsp:8[Rd](注 1)	5
	B/W/L	size	dsp:16[Rs](注 1)	dsp:16[Rd](注 1)	6
	B/W/L	size	Rs	[Rd+]	3
	B/W/L	size	Rs	[-Rd]	3
	B/W/L	L	[Rs+]	Rd	3
	B/W/L	L	[-Rs]	Rd	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MOV.L #0, R2



MOV.L #128:8, R2 MOV.L #-128:8, R2 MOV.L R1, R2 MOV.L #0, [R2] MOV.W [R1], R2 MOV.W [R1, R2], R3 MOV.W [R1, R2], R3 MOV.W R1, [R2, R3] MOV.W [R1], [R2] MOV.B R1, [R2+] MOV.B R1, [-R2] MOV.B R1, [-R2] MOV.B [-R1], R2



MOVU

Transfer unsigned data

MOVU

MOVe Unsigned data

[Syntax] MOVU.size src, dest

[Operation] dest = src;

[Function]

- This instruction transfers src to dest as listed in the following table.

[Instruction Format]

Syntax	size	Pro- cessng Size	src	dest	Code size (Byte)
MOVU.size src, dest	B/W	L	dsp:5[Rs](注 1) (Rs=R0 ~ R7)	Rd (Rd=R0 ~ R7)	2
	B/W	L	Rs	Rd	2
	B/W	L	[Rs]	Rd	2
	B/W	L	dsp:8[Rs](注 1)	Rd	3
	B/W	L	dsp:16[Rs](注 1)	Rd	4
	B/W	L	[Ri, Rb]	Rd	3
	B/W	L	[Rs+]	Rd	3
	B/W	L	[-Rs]	Rd	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MOVU.W 2[R1], R2 MOVU.W R1, R2 MOVU.B [R1+], R2 MOVU.B [-R1], R2


MUL

Multiplication

MUL

MULtiply

[Syntax] (1)MUL src, dest (2)MUL src, src2, dest

[Operation] (1)dest = src * dest; (2)dest = src * src2;

[Function]

- (1)This instruction multiplies src and dest and places the result in dest.

- The calculation is performed in 32 bits and the lower-order 32 bits of the result are placed.
- The operation result will be the same whether a singed or unsigned multiply is executed.
- (2)This instruction multiplies src and src2 and places the result in dest.
 - The calculation is performed in 32 bits and the lower-order 32 bits of the result are placed.
 - The operation result will be the same whether a singed or unsigned multiply is executed.
- Note The accumulator (ACC) is used to perform the function. The value of ACC after executing the instruction is undefined.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	dest	Code size (Byte)
(1)MUL src, dest	L	#UIMM:4	-	Rd	2
	L	#SIMM:8	-	Rd	3
	L	#SIMM:16	-	Rd	4
	L	#SIMM:24	-	Rd	5
	L	#IMM:32	-	Rd	6
	L	Rs	-	Rd	2
	L	[Rs].memex	-	Rd	2 (memex == UB) 3 (memex != UB)
	L	dsp:8[Rs].memex(注)	-	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:16[Rs].memex(注)	-	Rd	4 (memex == UB) 5 (memex != UB)
	L	Rs	Rs2	Rd	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example]

 MUL
 #10, R2

 MUL
 R1, R2

 MUL
 [R1], R2

 MUL
 4[R1].W, R2

 MUL
 R1, R2, R3



MULHI

Multiply the high-order word

MULHI

MULtiply HIgh-order word

[Syntax] MULHI src, src2

[Operation] signed short tmp1, tmp2; signed long long tmp3; tmp1 = (signed short) (src >> 16); tmp2 = (signed short) (src2 >> 16); tmp3 = (signed long) tmp1 * (signed long) tmp2; ACC = (tmp3 << 16);

[Function]

- This instruction multiplies the higher-order 16 bits of src by the higher-order 16 bits of src2, and stores the result in the accumulator (ACC). When the result is stored, the least significant bit of the result corresponds to bit 16 of ACC, and the section corresponding to bits 63 to 48 of ACC is sign-extended. Moreover, bits 15 to 0 of ACC are cleared to 0. The higher-order 16 bits of src and the higher-order 16 bits of src2 are treated as signed integers.

[Instruction Format]

Syntax	src	src2	Code size (Byte)
MULHI src, src2	Rs	Rs2	3

[Flag Change]

[Description Example] MULHI R1, R2



⁻ This instruction does not affect the states of flags.

MULLO

Multiply the low-order word

MULLO

MULtiply LOw-order word

[Syntax] MULLO src, src2

[Operation] signed short tmp1, tmp2; signed long long tmp3; tmp1 = (signed short) src; tmp2 = (signed short) src2; tmp3 = (signed long) tmp1 * (signed long) tmp2; ACC = (tmp3 << 16);

[Function]

- This instruction multiplies the lower-order 16 bits of src by the lower-order 16 bits of src2, and stores the result in the accumulator (ACC). When the result is stored, the least significant bit of the result corresponds to bit 16 of ACC, and the section corresponding to bits 63 to 48 of ACC is sign-extended. Moreover, bits 15 to 0 of ACC are cleared to 0. The lower-order 16 bits of src and the lower-order 16 bits of src2 are treated as signed integers.

[Instruction Format]

Syntax	src	src2	Code size (Byte)
MULLO src, src2	Rs	Rd	3

[Flag Change]

[Description Example] MULLO R1, R2



⁻ This instruction does not affect the states of flags.

MVFACHI

Move the high-order longword MVFACHI from accumulator

MoVe From ACcumulator Highorder longword

[Syntax] MVFACHI dest

[Operation] dest = (signed long) (ACC >> 32);

[Function]

- This instruction moves the higher-order 32 bits of the accumulator (ACC) to dest.

[Instruction Format]

Syntax	dest	Code size (Byte)
MVFACHI dest	Rd	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MVFACHI R1

R20UT2999EJ0100 Rev.1.00 May. 20, 2014



MVFACMI

Move the middle-order longword MVFACMI from accumulator

MoVe From ACcumulator MIddleorder longword

[Syntax] MVFACMI dest

[Operation] dest = (signed long) (ACC >> 16);

[Function]

- This instruction moves the contents of bits 47 to 16 of the accumulator (ACC) to dest.

[Instruction Format]

Syntax	dest	Code size (Byte)
MVFACMI dest	Rd	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MVFACMI R1



MVFC

Transfer from a control register MVFC

MoVe From Control register

[Syntax] MVFC src, dest

[Operation] dest = src;

[Function]

- This instruction transfers src to dest.

- When the PC is specified as src, this instruction pushes its own address onto the stack.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
MVFC src, dest	L	Rx(注)	Rd	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MVFC USP, R1



MVTACHI

Move the high-order longword MVTACHI to accumulator MoVe To ACcumulator HIgh-order longword

[Syntax] MVTACHI src

[Operation]

ACC = (ACC & 0000000FFFFFFFh) | ((signed long long)src << 32);

[Function]

- This instruction moves the contents of src to the higher-order 32 bits (bits 63 to 32) of the accumulator (ACC).

[Instruction Format]

Syntax	SrC	Code size (Byte)
MVTACHI src	Rs	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MVTACHI R1



MVTACLO

Move the low-order longword MVTACLO to accumulator

MoVe To ACcumulator LOw-order longword

[Syntax] MVTACLO src

[Operation] ACC = (ACC & FFFFFFF0000000h) | src;

[Function]

- This instruction moves the contents of src to the lower-order 32 bits (bits 31 to 0) of the accumulator (ACC).

[Instruction Format]

Syntax	SrC	Code size (Byte)
MVTACLO src	Rs	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MVTACLO R1



MVTC

Transfer to a control register

MVTC

MoVe To Control register

[Syntax] MVTC src, dest

[Operation] dest = src;

[Function]

- This instruction transfers src to dest.
- In user mode, writing to the ISP, INTB, BPC, BPSW, and FINTV, and the IPL[3:0], PM, U, and I bits in the PSW is ignored. In supervisor mode, writing to the PM bit in the PSW is ignored.

[Instruction Format] 【命令フォーマット】

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
MVTC src, dest	L	#SIMM:8	Rx(注)	7
	L	#SIMM:16	Rx(注)	3
	L	#SIMM:24	Rx(注)	3
	L	#IMM:32	Rx(注)	4
	L	Rs	Rx(注)	5

[Flag Change]

Flag	С	Z	S	0
Change	*	*	*	*

Note *The flag changes only when dest is the PSW. [Description Example]

MVTC #0FFFFF000h, INTB MVTC R1, USP



MVTIPL

Interrupt priority level setting MVTIPL

MoVe To Interrupt Priority Level

[Syntax] MVTIPL src

[Operation] IPL = src; [Function]

- This instruction transfers src to the IPL[3:0] bits in the PSW.
- This instruction is a privileged instruction. Attempting to execute this instruction in user mode generates a privileged instruction exception.
- The value of src is an unsigned integer in the range 0 ? src ? 15.

[Instruction Format]

Syntax	src	Code size (Byte)
MVTIPL src	#IMM:32	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] MVTIPL #2



NEG

Two's complementation

NEG

NEGate

[Syntax] (1)NEG dest (2)NEG src, dest

[Operation] (1)dest = -dest; (2)dest = -src;

[Function]

- (1)This instruction arithmetically inverts (takes the two's complement of) dest and places the result in dest.

- (2)This instruction arithmetically inverts (takes the two's complement of) src and places the result in dest.

[Instruction Format]

Sy	/ntax	Pro- cessng Size	src	dest	Code size (Byte)
(1)NEG	dest	L	_	Rd	2
(2)NEG	src, dest	L	Rs	Rd	3

[Flag Change]

	Flag		С	Z	S	0
0	Change	0	0			

Conditions

- C : The flag is set if dest is 0 after the operation; otherwise it is cleared.
- Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.
- S : The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.
- O : (1) The flag is set if dest before the operation was 80000000h; otherwise it is cleared.

(2) The flag is set if src before the operation was 80000000h; otherwise it is cleared.

[Description Example]

NEG R1 NEG R1, R2



NOP

No operation

NOP

No OPeration

[Syntax] NOP

[Operation] /* No operation */

[Function]

- This instruction executes no process. The operation will be continued from the next instruction.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
(1)NOT dest	L	-	Rd	2
(2)NOT src, dest	L	Rs	Rd	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] NOP



NOT

Logical complementation

NOT

NOT

[Syntax] (1)NOT dest (2)NOT src, dest

[Operation] (1)dest = ~dest; (2)dest = ~src;

[Function]

- (1)This instruction logically inverts dest and places the result in dest.

- (2)This instruction logically inverts src and places the result in dest.

[Instruction Format]

Si	yntax	Pro- cessng Size	src	dest	Code size (Byte)
(1)NOT	dest	L	-	Rd	2
(2)NOT	src, dest	L	Rs	Rd	3

[Flag Change]

Flag	С	Z	S	0
Char@e 🔿	_			_

Conditions

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S : The flag is set if the MSB of dest after the operation is 1 ; otherwise it is cleared.

[Description Example]

NOT R1 NOT R1, R2



OR

Logical OR

OR

OR

[Syntax] (1)OR src, dest (2)OR src, src2, dest

[Operation] (1)dest = dest | src; (2)dest = src | src2;

[Function]

- (1)This instruction takes the logical OR of dest and src and places the result in dest.

- (2)This instruction takes the logical OR of src and src2 and places the result in dest.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	dest	Code size (Byte)
(1)OR src, dest	L	#UIMM:4	_	Rd	2
	L	#SIMM:8	-	Rd	3
	L	#SIMM:16	-	Rd	4
	L	#SIMM:24	-	Rd	5
	L	#IMM:32	-	Rd	6
	L	Rs	-	Rd	2
	L	[Rs].memex	-	Rd	2 (memex == UB) 3 (memex != UB)
	L	dsp:8[Rs].memex(注)	_	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:16[Rs].memex (注)	-	Rd	4 (memex == UB) 5 (memex != UB)
(2)OR src, src2, dest	L	Rs	Rs2	Rd	3

[Flag Change]

Flag	С	Z	S	0
Char@e 🔿	_			_

Conditions

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S : The flag is set if the MSB of dest after the operation is 1 ; otherwise it is cleared.

[Description Example]

OR #8, R1

OR R1, R2



OR	[R1], R2
OR	8[R1].L, R2
OR	R1, R2, R3



POP

Restoring data from stack to reg- POP ister

POP data from the stack

[Syntax] POP dest

[Operation] tmp = *SP; SP = SP + 4; dest = tmp;

[Function]

- This instruction restores data from the stack and transfers it to dest.

- The stack pointer in use is specified by the U bit in the PSW.

[Instruction Format]

Syntax	Processng Size	dest	Code size (Byte)
POP dest	L	Rd	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example]

POP R1



POPC

Restoring a control register

POPC

POP Control register

[Syntax] POPC dest

[Operation] tmp = *SP; SP = SP + 4; dest = tmp;

[Function]

- This instruction restores data from the stack and transfers it to the control register specified as dest.

- The stack pointer in use is specified by the U bit in the PSW.

- In user mode, writing to the ISP, INTB, BPC, BPSW, and FINTV, and the IPL[3:0], PM, U, and I bits in the PSW is ignored. In supervisor mode, writing to the PM bit in the PSW is ignored.

[Instruction Format]

Syntax	Processng Size	dest	Code size (Byte)
POPC dest	L	Rx(注)	2

[Flag Change]

Flag	С	Z	S	0
Change	*	*	*	*

Note * The flag changes only when dest is the PSW.

[Description Example]

POPC PSW



POPM

Restoring multiple registers from POPM the stack

POP Multiple registers

[Syntax] POPM dest-dest2

```
[Operation]
signed char i;
for ( i = register_num(dest); i <= register_num(dest2); i++ ) {
    tmp = *SP;
    SP = SP + 4;
    register(i) = tmp;
}
```

[Function]

- This instruction restores values from the stack to the block of registers in the range specified by dest and dest2.
- The range is specified by first and last register numbers. Note that the condition (first register number < last register number) must be satisfied.
- R0 cannot be specified.
- The stack pointer in use is specified by the U bit in the PSW.
- Registers are restored from the stack from R1 to R15.

[Instruction Format]

Syntax	Pro- cessng Size	dest	dest2	Code size (Byte)
POPM dest-dest2	L	Rd (Rd=R1 ~ R14)	Rd2 (Rd2=R2 ~ R15)	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] POPM R1-R3 POPM R4-R8



PUSH

Saving data on the stack

PUSH

PUSH data onto the stack

[Syntax] PUSH.size src

[Operation] tmp = src; SP = SP - 4 *; *SP = tmp;

* SP is always decremented by 4 even when the size specifier (.size) is .B or .W. The higher-order 24 and 16 bits in the respective cases (.B and .W) are undefined.

[Function]

Note

- This instruction pushes src onto the stack.
- When src is in register and the size specifier for the PUSH instruction is .B or .W, the byte or word of data from the LSB in the register are saved respectively.
- The transfer to the stack is processed in longwords. When the size specifier is .B or .W, the higher-order 24 or 16 bits are undefined respectively.
- The stack pointer in use is specified by the U bit in the PSW.

[Instruction Format]

Syntax	size	Processng Size	src	Code size (Byte)
PUSH.size src	B/W/L	L	Rs	2
	B/W/L	L	[Rs]	2
	B/W/L	L	dsp:8[Rs](注)	3
	B/W/L	L	dsp:16[Rs](注)	4

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] PUSH.B R1 PUSH.L [R1]



PUSHC

Saving a control register PUSH Control register

PUSHC

[Syntax] PUSHC src

[Operation] tmp = src; SP = SP - 4; *SP = tmp;

[Function]

- This instruction pushes the control register specified by src onto the stack.

- The stack pointer in use is specified by the U bit in the PSW.

- When the PC is specified as src, this instruction pushes its own address onto the stack..

[Instruction Format]

Syntax	Processng Size	src	Code size (Byte)
PUSHC src	L	Rx(注)	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example]

PUSHC PSW



PUSHM

Saving multiple registers

PUSHM

PUSH Multiple registers

[Syntax] PUSHM src-src2

```
[Operation]
signed char i;
for ( i = register_num(src2); i >= register_num(src); i-- ) {
    tmp = register(i);
    SP = SP - 4;
    *SP = tmp;
}
```

[Function]

- This instruction saves values to the stack from the block of registers in the range specified by src and src2.
- The range is specified by first and last register numbers. Note that the condition (first register number < last register number) must be satisfied.
- R0 cannot be specified.
- The stack pointer in use is specified by the U bit in the PSW.
- Registers are saved in the stack from R15 to R1.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	Code size (Byte)
PUSHM src-src2	L	Rs (Rs=R1 ~ R14)	Rs2 (Rs2=R2 ~ R15)	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] PUSHM R1-R3 PUSHM R4-R8



RACW

Round the accumulator word RACW

Round ACcumulator Word

[Syntax] RACW src

[Operation] signed long long tmp; tmp = (signed long long) ACC << src; tmp = tmp + 000000080000000h; if (tmp > (signed long long) 00007FFF00000000h) ACC = 00007FFF00000000h; else if (tmp < (signed long long) FFFF80000000000h) ACC = FFFF80000000000h; else ACC = tmp & FFFFFFF00000000h

[Function]

- This instruction rounds the value of the accumulator into a word and stores the result in the accumulator.

- The RACW instruction is executed according to the following procedures.
- Processing 1:
- The value of the accumulator is shifted to the left by one or two bits as specified by src
- Processing 2:
- The value of the accumulator changes according to the value of 64 bits after the contents have been shifted to the left by one or two bits.

[Instruction Format]

Syntax	SrC	Code size (Byte)
RACW src	#IMM:1(注) (IMM:1 = 1 ~ 2)	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example]

RACW #1 RACW #2



REVL

Endian conversion

REVL

REVerse Longword data

[Syntax] REVL src, dest

[Operation] Rd = { Rs[7:0], Rs[15:8], Rs[23:16], Rs[31:24] }

[Function]

- This instruction converts the endian byte order within a 32-bit datum, which is specified by src, and saves the result in dest.

[Instruction Format]

Syntax	src	dest	Code size (Byte)
REVL src, dest	Rs	Rd	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] REVL R1, R2



REVW

Endian conversion

REVW

REVerse Word data

[Syntax] REVW src, dest

[Operation] Rd = { Rs[23:16], Rs[31:24], Rs[7:0], Rs[15:8] }

[Function]

- This instruction converts the endian byte order within the higher- and lower-order 16-bit data, which are specified by src, and saves the result in dest.

[Instruction Format]

Syntax	src	dest	Code size (Byte)
REVW src, dest	Rs	Rd	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] REVW R1, R2



RMPA	Multiply-and-accumulate ope tion	era- RMPA
	Repeated MultiPly and Accul late	mu-
[Syntax] RMPA.size		
[Operation] while (R3 != R6:R5:R4 = R1 = R1 + r R2 = R2 + r R3 = R3 - 1 }	0) { = R6:R5:R4 + *R1 * *R2; n; ;	
Notes 1.	1. If this instruction is executed with R3 set to 0, it is ignored	and has no effect on registers and flags.
Notes 2.	2. When the size specifier (.size) is .B, .W, or .L, n is 1, 2, or	r 4, respectively.

[Function]

- This instruction performs a multiply-and-accumulate operation with the multiplicand addresses specified by R1, the multiplier addresses specified by R2, and the number of multiply-and-accumulate operations specified by R3. The operands and result are handled as signed values, and the result is placed in R6:R5:R4 as an 80-bit datum. Note that the higher-order 16 bits of R6 are set to the value obtained by sign-extending the lower-order 16 bits of R6.
- The greatest value that is specifiable in R3 is 00010000h
- The data in R1 and R2 are undefined when instruction execution is completed.
- Specify the initial value in R6:R5:R4 before executing the instruction. Furthermore, be sure to set R6 to FFFFFFFh when R5:R4 is negative or to 00000000h if R5:R4 is positive.
- An interrupt request during execution of this instruction will be accepted, so processing of the instruction will be suspended. That is, execution of the instruction will continue on return from the interrupt processing routine. However, be sure to save the contents of the R1, R2, R3, R4, R5, R6, and PSW when an interrupt is generated and restore them when execution is returned from the interrupt routine.
- In execution of the instruction, the data may be prefetched from the multiplicand addresses specified by R1 and the multiplier addresses specified by R2, with R3 as the upper limit. For details of the data size to be prefetched, refer to the hardware manual of each product.
- Note The accumulator (ACC) is used to perform the function. The value of ACC after executing the instruction is undefined.

[Instruction Format]

Syntax	size	size	Code size (Byte)
RMPA.size	B/W/L	size	2

[Flag Change]

Flag	С	Z	S	0
O Change	_	_		

Conditions

S : The flag is set if the MSB of R6 is 1; otherwise it is cleared.

O: The flag is set if the R6:R5:R4 data is greater than 2**53-1 or smaller than -2**53; otherwise it is cleared.

[Description Example]

RMPA.W

ROLC

Rotation with carry to left

ROLC

ROtate Left with Carry

[Syntax] ROLC dest

[Operation] dest <<= 1; if (C == 0) { dest &= FFFFFFEh; } else { dest |= 00000001h; }

[Function]

- This instruction treats dest and the C flag as a unit, rotating the whole one bit to the left.

[Instruction Format]

Syntax	Processng Size	dest	Code size (Byte)
ROLC dest	L	Rd	2

[Flag Change]

Flag	С	Z	S	0
Char@e	0			-

Conditions

C: The flag is set if the shifted-out bit is 1; otherwise it is cleared.

Z: The flag is set if dest is 0 after the operation; otherwise it is cleared.

S: The flag is set if the MSB of dest after the operation is 1 ; otherwise it is cleared.

[Description Example]

ROLC R1



RORC

Rotation with carry to right

RORC

ROtate Right with Carry

[Syntax] RORC dest

[Operation] dest >>= 1; if (C == 0) { dest &= 7FFFFFFh; } else { dest |= 8000000h; }

[Function]

- This instruction treats dest and the C flag as a unit, rotating the whole one bit to the right.

[Instruction Format]

Syntax	Processng Size	dest	Code size (Byte)
RORC dest	L	Rd	2

[Flag Change]

Flag	С	Z	S	0
Chan@e C	0			-

Conditions

- C: The flag is set if the shifted-out bit is 1; otherwise it is cleared.
- Z: The flag is set if dest is 0 after the operation; otherwise it is cleared.
- S: The flag is set if the MSB of dest after the operation is 1 ; otherwise it is cleared.

[Description Example]

RORC R1



ROTL

Rotation to left

ROTL

ROTate Left

[Syntax] ROTL src, dest

[Operation] unsigned long tmp0, tmp1; tmp0 = src & 31; tmp1 = dest << tmp0; dest = ((unsigned long) dest >> (32 - tmp0)) | tmp1; [Function]

- This instruction rotates dest leftward by the number of bit positions specified by src and saves the value in dest. Bits overflowing from the MSB are transferred to the LSB and to the C flag.
- src is an unsigned integer in the range of 0 ? src ? 31.

- When src is in register, only five bits in the LSB are valid.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
ROTL src, dest	L	#IMM:5	Rd	3
	L	Rs	Rd	3

[Flag Change]

Flag	С	Z	S	0
Change 🔿	0			-

Conditions

C: After the operation, this flag will have the same LSB value as dest. In addition, when src is 0, this flag will have the same LSB value as dest.

Z: The flag is set if dest is 0 after the operation; otherwise it is cleared.

S: The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.

[Description Example]

ROTL #1, R1

ROTL R1, R2]



ROTR

Rotation to right

ROTR

ROTate Right

[Syntax] ROTR src, dest

[Operation] unsigned long tmp0, tmp1; tmp0 = src & 31; tmp1 = (unsigned long) dest >> tmp0; dest = (dest << (32 - tmp0)) | tmp1;

[Function]

- This instruction rotates dest rightward by the number of bit positions specified by src and saves the value in dest. Bits overflowing from the LSB are transferred to the MSB and to the C flag.

- src is an unsigned integer in the range of 0 ? src ? 31.

- When src is in register, only five bits in the LSB are valid.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
ROTR src, dest	L	#IMM:5	Rd	3
	L	Rs	Rd	3

[Flag Change]

Flag		С	Z	S	0
Change	0	0			_

Conditions

C: After the operation, this flag will have the same MSB value as dest. In addition, when src is 0, this flag will have the same MSB value as dest.

Z: The flag is set if dest is 0 after the operation; otherwise it is cleared.

S: The flag is set if the MSB of dest after the operation is 1 ; otherwise it is cleared.

[Description Example]

ROTR #1, R1 ROTR R1, R2

ROTR R1, R2



ROUND

Conversion from floating-point to ROUND integer

ROUND floating-point to integer

[Syntax] ROUND src, dest

[Operation] dest = (signed long) src;

[Function]

- This instruction converts the single-precision floating-point number stored in src into a signed longword (32-bit) integer and places the result in dest. The result is rounded according to the setting of the RM[1:0] bits in the FPSW.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
ROUND src, dest	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L(注)	Rd	4
	L	dsp:16[Rs].L(注)	Rd	4

[Flag Change]

	Flag	С	Z	S	0	CV	CZ	CU	СХ	CE	FV	FO	FZ	FU	FX
0	Change 🔾	ð	0	\bigcirc	ð	0	\bigcirc	0		0	0				

Conditions

- Z : The flag is set if the result of the operation is 0; otherwise it is cleared.
- S :The flag is set if the sign bit (bit 31) of the result of the operation is 1; otherwise it is cleared.
- CV : The flag is set if an invalid operation exception is generated; otherwise it is cleared.
- CO : The value of the flag is always 0.
- CZ : The value of the flag is always 0.
- CU : The value of the flag is always 0.
- CX :The flag is set if an inexact exception is generated; otherwise it is cleared.
- CE :The flag is set if an unimplemented processing exception is generated; otherwise it is cleared...
- FV :The flag is set if an invalid operation exception is generated; otherwise it does not change.
- FX:The flag is set if an inexact exception is generated; otherwise it does not change.

Note

The FX and FV flags do not change if any of the exception enable bits EX and EV is 1. The S and Z flags do not change when an exception is generated.

[Description Example] ROUND R1, R2

ROUND [R1], R2



RTE

Return from the exception

RTE

ReTurn from Exception

[Syntax] RŤE

[Operation] PC = *SP; SP = SP + 4;tmp = *SP; SP = SP + 4;PSW = tmp;

[Function]

- This instruction returns execution from the exception handling routine by restoring the PC and PSW contents that were preserved when the exception was accepted.
- This instruction is a privileged instruction. Attempting to execute this instruction in user mode generates a privileged instruction exception.
- If returning is accompanied by a transition to user mode, the U bit in the PSW becomes 1.

[Instruction Format]

Syntax	Code size (Byte)
RTE	2

[Flag Change]

Flag	С	V	S	0
Change	*	*	*	*

* The flags become the corresponding values on the stack. Note

[Description Example]

RTE



RTFI

Return from the fast interrupt RTFI

ReTurn from Fast Interrupt

[Syntax] RTFI

[Operation] PSW = BPSW; PC = BPC;

[Function]

- This instruction returns execution from the fast-interrupt handler by restoring the PC and PSW contents that were saved in the BPC and BPSW when the fast interrupt request was accepted.
- This instruction is a privileged instruction. Attempting to execute this instruction in user mode generates a privileged instruction exception.
- If returning is accompanied by a transition to user mode, the U bit in the PSW becomes 1.
- The data in the BPC and BPSW are undefined when instruction execution is completed.

[Instruction Format]

Syntax	Code size (Byte)
RTFI	2

[Flag Change]

Flag	С	V	S	0
Change	*	*	*	*

Note * The flags become the corresponding values from the BPSW.

[Description Example] RTFI



RTS

Returning from a subroutine RTS

ReTurn from Subroutine

[Syntax] RTS

[Operation] PC = *SP; SP = SP + 4;

[Function]

- This instruction rotates *dest* one bit to the left including the C flag.

[Instruction Format]

Syntax	Code size (Byte)
RTS	1

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example]

RTS



RTSD	Releasing stack frame and return- ing from subroutine	RTSD
	ReTurn from Subroutine and Deallocate stack frame	
[Syntax] (1)RTSD src (2)RTSD src, dest-dest2		
[Operation] (1)SP = SP + src; PC = *SP; SP = SP + 4;		
<pre>(2)signed char i; SP = SP + (src - (register_num(de for (i = register_num(dest); i <= register tmp = *SP; SP = SP + 4; register(i) = tmp; } PC = *SP; SP = SP + 4;</pre>	est2) - register_num(dest) +1) * 4); gister_num(dest2); i++) {	

[Function]

- This instruction rotates *dest* one bit to the right including the C flag.
- This instruction restores values for the block of registers in the range specified by dest and dest2 from the stack.
- The range is specified by first and last register numbers. Note that the condition (first register number ? last register number) must be satisfied.
- R0 cannot be specified.
- The stack pointer in use is specified by the U bit in the PSW.
- Registers are restored from the stack from R1 to R15:

[Instruction Format]

Syntax	src	dest	dest2	Code size (Byte)
(1)RTSD src	#UIMM:8 (Note)	_	_	2
(2)RTSD src, dest- dest2	#UIMM:8 ^(Note)	Rd (Rd=R1~R15)	Rd2 (Rd=R1~R15)	3

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] RTSD #4 RTSD #16, R5-R7



SAT

Saturation of signed 32-bit data SAT SATurate signed 32-bit data

```
[Syntax]
SAT dest
[Operation]
if ( O == 1 && S == 1 )
dest = 7FFFFFFh;
else if ( O == 1 && S == 0 )
dest = 80000000h;
```

[Function]

- This instruction rotates *dest* left or right the number of bits indicated by *src*. Bits overflowing from LSB (MSB) are transferred to MSB (LSB) and the C flag.
- The direction of rotation is determined by the sign of *src*. If *src* is positive, bits are rotated left; if negative, bits are rotated right.
- If *src* is an immediate value, the number of bits rotated is -8 to -1 or +1 to +8. Values less than -8, equal to 0, or greater than +8 are not valid.
- If *src* is a register and (.B) is selected as the size specifier (.size), the number of bits rotated is –8 to +8. Although a value of 0 may be set, no bits are rotated and no flags are changed. If a value less than –8 or greater than +8 is set, the result of the rotation is undefined.
- If *src* is a register and (.W) is selected as the size specifier (.size), the number of bits rotated is –16 to +16. Although a value of 0 may be set, no bits are rotated and no flags are changed. If a value less than –16 or greater than +16 is set, the result of the rotation is undefined.

[Instruction Format]

Syntax	Processng Size	dest	Code size (Byte)
SAT dest	L	Rd	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] SAT R1



SATR

Saturation of signed 64-bit data SATR for RMPA

SATuRate signed 64-bit data for RMPA

[Syntax] SATR

[Operation] if (O == 1 && S == 0) R6:R5:R4 = 00000007FFFFFFFFFFFFF; else if (O == 1 && S == 1) R6:R5:R4 = FFFFFF8000000000000000;

[Function]

- This instruction performs a 64-bit signed saturation operation.

- When the O flag is 1 and the S flag is 0, the result of the operation is 00000007FFFFFFFFFFFFFFFFFF and it is placed in R6:R5:R4. When the O flag is 1 and the S flag is 1, the result of the operation is FFFFFFF800000000000000 and it is place in R6:R5:R4. In other cases, the R6:R5:R4 value does not change.

[Instruction Format]

Syntax	Code size (Byte)
SATR	2

[Flag Change]

- This instruction does not affect the states of flags. [Description Example]

SATR


SBB

Subtraction with borrow

SBB

SuBtract with Borrow

[Syntax] SBB src, dest

[Operation] dest = dest - src - !C;

[Function]

- This instruction subtracts src and the inverse of the C flag (borrow) from dest and places the result in dest.

[Instruction Format]

	Syntax	Pro- cessng Size	src	dest	Code size (Byte)
SBB	src, dest	L	Rs	Rd	3
		L	[Rs].L	Rd	4
		L	dsp:8[Rs].L(注)	Rd	5
		L	dsp:16[Rs].L(注)	Rd	6

[Flag Change]

	Flag		С	Z	S	0
0	Change	0	0			

Conditions

C : The flag is set if an unsigned operation produces no overflow; otherwise it is cleared.

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S : The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.

O : The flag is set if a signed operation produces an overflow; otherwise it is cleared.

[Description Example]

SBB R1, R2

SBB [R1], R2



SCCnd

Condition setting

SCCnd

Store Condition Conditionally

[Syntax] SCCnd.size dest

[Operation] if (Cnd) dest = 1; else dest = 0;

[Function]

- This instruction moves the truth-value of the condition specified by Cnd to dest; that is, 1 or 0 is stored to dest if the condition is true or false, respectively.

- The following table lists the types of SCCnd.

BCnd		Condition	Expression
SCGEU, SCC	C == 1	Equal to or greater than/C flag is 1	<=
SCEQ, SCZ	Z == 1	Equal to/Z flag is 1	=
SCGTU	(C & ~Z) == 1	Greater than	<
SCPZ	S == 0	Positive or zero	> 0
SCGE	(S ^ O) == 0	Equal to or greater than as signed integer	<=
SCGT	((S ^ O) Z) == 0	Greater than as signed integer	<
SCO	O == 1	O flag is 1	
SCLTU, SCNC	C == 0	Less than/C flag is 0	<=
SCNE, SCNZ	Z == 0	Not equal to/Z flag is 0	
SCLEU	(C & ~Z) == 0	Equal to or less than	
SCN	S == 1	Negative	0>
SCLE	((S ^ O) Z) == 1	Equal to or less than as signed integer	>=
SCLT	(S ^ O) == 1	Less than as signed integer	>
SCNO	O == 0	O flag is 0	

[Instruction Format]

Syntax	size	Processng Size	dest	Code size (Byte)
SCCnd.size dest	L	L	Rd	3
	B/W/L	size	[Rd]	3



Syntax	size	Processng Size	dest	Code size (Byte)
	B/W/L	size	dsp:8[Rd](注)	4
	B/W/L	size	dsp:16[Rd](注)	5

[Flag Change]

- This instruction does not affect the states of flags. [Description Example]

SCC.L R2

SCNE.W [R2]



SCMPU

String comparison

SCMPU

String CoMPare Until not equal

[Syntax] SCMPU

```
[Operation]
unsigned char *R2, *R1, tmp0, tmp1;
unsigned long R3;
while ( R3 != 0 ) {
    tmp0 = *R1++;
    tmp1 = *R2++;
    R3--;
    if ( tmp0 != tmp1 || tmp0 == '\0' ) {
        break;
    }
}
```

Note

If this instruction is executed with R3 set to 0, it is ignored and has no effect on registers and flags.

[Function]

- This instruction compares strings in successively higher addresses specified by R1, which indicates the source address for comparision, and R2, which indicates the destination address for comparision, until the values do not match or the null character "\0" (= 00h) is detected, with the number of bytes specified by R3 as the upper limit.
- In execution of the instruction, the data may be prefetched from the source address for comparison specified by R1 and the destination address for comparison specified by R2, with R3 as the upper limit. For details of the data size to be prefetched, refer to the hardware manual of each product.
- The contents of R1 and R2 are undefined upon completion of the instruction.
- An interrupt request during execution of this instruction will be accepted, so processing of the instruction will be suspended. That is, execution of the instruction will continue on return from the interrupt processing routine. However, be sure to save the contents of the R1, R2, R3, and PSW when an interrupt is generated and restore them when execution is returned from the interrupt routine.

[Instruction Format]

Syntax	Processng Size	Code size (Byte)
SCMPU	В	2

[Flag Change]

Flag	С	Z	S	0
Change C	0		_	_

Conditions

C : This flag is set if the operation of (*R1 - *R2) as unsigned integers produces a value greater than or equal to 0; otherwise it is cleared.

Z : This flag is set if the two strings have matched; otherwise it is cleared.

[Description Example] SCMPU



SETPSW

Setting a flag or bit in the PSW **SETPSW**

SET flag of PSW

LSB

LSB 🖣

[Syntax] SETPSW dest

[Operation] dest = 1;

When src < 0

When src > 0

[Function]

- This instruction clears the O, S, Z, or C flag, which is specified by dest, or the U or I bit.

dest

dest

- In user mode, writing to the U or I bit in the PSW will be ignored. In supervisor mode, all flags and bits can be written to.

► C

- 0

[Instruction Format]

Syntax	dest	Code size (Byte)
SETPSW dest	flag	2

[Flag Change]

Flag	С	Z	S	0
Change	*	*	*	*

Note * The specified flag is set to 1.

→ MSB

C MSB

0 -

[Description Example] SETPSW C SETPSW Z



SHAR

Arithmetic shift to the right

SHAR

SHift Arithmetic Right

```
[Syntax]
(1)SHAR src, dest
(2)SHAR src, src2, dest
```

```
[Operation]
(1)dest = ( signed long ) dest >> ( src & 31 );
(2)dest = ( signed long ) src2 >> ( src & 31 );
```

[Function]

- (1)This instruction arithmetically shifts dest to the right by the number of bit positions specified by src and saves the value in dest.
 - Bits overflowing from the LSB are transferred to the C flag.
 - src is an unsigned in the range of 0 ? src ? 31.
 - When src is in register, only five bits in the LSB are valid.
- (2)After this instruction transfers src2 to dest, it arithmetically shifts dest to the right by the number of bit positions specified by src and saves the value in dest.
 - Bits overflowing from the LSB are transferred to the C flag.
 - src is an unsigned integer in the range of 0 ? src ? 31.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	dest	Code size (Byte)
(1)SHAR src, dest	L	#IMM:5	-	Rd	2
	L	Rs	-	Rd	3
(1)SHAR src, src2, dest	L	#IMM:5	Rs	Rd	3

[Flag Change]

	Flag		С	Z	S	0
0	Change	0	0			

Conditions

C : The flag is set if the shifted-out bit is 1; otherwise it is cleared. However, when src is 0, this flag is also cleared.

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S : The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.

O : The flag is cleared to 0.

[Description Example]

 SHAR
 #3, R2

 SHAR
 R1, R2

 SHAR
 #3, R1, R2



SHLL

Logical and arithmetic shift to the SHLL left

SHift Logical and arithmetic Left

[Syntax] (1)SHLL src, dest (2)SHLL src, src2, dest

[Operation] (1)dest = dest << (src & 31); (2)dest = src2 << (src & 31);

[Function]

- (1)This instruction arithmetically shifts dest to the left by the number of bit positions specified by src and saves the value in dest.
 - Bits overflowing from the MSB are transferred to the C flag.
 - When src is in register, only five bits in the LSB are valid.
 - src is an unsigned integer in the range of 0 ? src ? 31.
- (2)After this instruction transfers src2 to dest, it arithmetically shifts dest to the left by the number of bit positions specified by src and saves the value in dest.
 - Bits overflowing from the MSB are transferred to the C flag.
 - src is an unsigned integer in the range of 0 ? src ? 31.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	dest	Code size (Byte)
(1)SHLL src, dest	L	#IMM:5	-	Rd	2
	L	Rs	-	Rd	3
(1)SHLL src, src2, dest	L	#IMM:5	Rs	Rd	3

[Flag Change]

Flag		С	Z	S	0	
0	Charge	0	0			

Conditions

C : The flag is set if the shifted-out bit is 1; otherwise it is cleared. However, when src is 0, this flag is also cleared.

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S : The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.

O : This bit is cleared to 0 when the MSB of the result of the operation is equal to all bit values that have been shifted out (i.e. the shift operation has not changed the sign); otherwise it is set to 1. However, when scr is 0, this flag is also cleared.

[Description Example]

SHLL #3, R2 SHLL R1, R2

SHLL R1, R2 SHLL #3, R1, R2

May. 20, 2014



SHLR

Logical shift to the right

SHLR

SHift Logical Right

[Syntax] (1)SHLR src, dest (2)SHLR src, src2, dest

[Operation]

(1)dest = (unsigned long) dest >> (src & 31); (2)dest = (unsigned long) src2 >> (src & 31);

[Function]

- (1)This instruction logically shifts dest to the right by the number of bit positions specified by src and saves the value in dest.
 - Bits overflowing from the LSB are transferred to the C flag.
 - src is an unsigned integer in the range of 0 ? src ? 31.
 - When src is in register, only five bits in the LSB are valid.
- (2)After this instruction transfers src2 to dest, it logically shifts dest to the right by the number of bit positions specified by src and saves the value in dest.
 - Bits overflowing from the LSB are transferred to the C flag.
 - src is an unsigned integer in the range of 0 ? src ? 31.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	dest	Code size (Byte)
(1)SHLR src, dest	L	#IMM:5	-	Rd	2
	L	Rs	-	Rd	3
(1)SHLR src, src2, dest	L	#IMM:5	Rs	Rd	3

[Flag Change]

Flag		С	Z	S	0	
0	Change	0	0			

Conditions

C : The flag is set if the shifted-out bit is 1; otherwise it is cleared. However, when src is 0, this flag is also cleared.

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S : The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.

[Description Example]

SHLR #3, R2

SHLR R1, R2

SHLR #3, R1, R2



SMOVB

Transferring a string backward SMOVB

Strings MOVe Backward

```
[Syntax]
SMOVB
```

```
[Operation]
unsigned char *R1, *R2;
unsigned long R3;
while ( R3 != 0 ) {
 *R1-- = *R2--;
R3 = R3 - 1;
}
```

Note If this instruction is executed with R3 set to 0, it is ignored and has no effect on registers and flags. [Function]

- This instruction transfers a string consisting of the number of bytes specified by R3 from the source address specified by R2 to the destination address specified by R1, with transfer proceeding in the direction of decreasing addresses.
- In execution of the instruction, data may be prefetched from the source address specified by R2, with R3 as the upper limit. For details of the data size to be prefetched, refer to the hardware manual of each product.
- The destination address specified by R1 should not be included in the range of data to be prefetched, which starts from the source address specified by R2.
- On completion of instruction execution, R1 and R2 indicate the next addresses in sequence from those for the last transfer.
- An interrupt request during execution of this instruction will be accepted, so processing of the instruction will be suspended. That is, execution of the instruction will continue on return from the interrupt processing routine. However, be sure to save the contents of the R1, R2, R3, and PSW when an interrupt is generated and restore them when execution is returned from the interrupt routine.

[Instruction Format]

Syntax	Processng Size	Code size (Byte)
SMOVB	В	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] SMOVB



SMOVF

Transferring a string forward SMOVF Strings MOVe Forward

[Syntax] SMOVF

```
[Operation]
unsigned char *R1, *R2;
unsigned long R3;
while ( R3 != 0 ) {
*R1++ = *R2++;
R3 = R3 - 1;
}
```

Note If this instruction is executed with R3 set to 0, it is ignored and has no effect on registers and flags. [Function]

- This instruction transfers a string consisting of the number of bytes specified by R3 from the source address specified by R2 to the destination address specified by R1, with transfer proceeding in the direction of increasing addresses.
- In execution of the instruction, data may be prefetched from the source address specified by R2, with R3 as the upper limit. For details of the data size to be prefetched, refer to the hardware manual of each product.
- The destination address specified by R1 should not be included in the range of data to be prefetched, which starts from the source address specified by R2.
- On completion of instruction execution, R1 and R2 indicate the next addresses in sequence from those for the last transfer.
- An interrupt request during execution of this instruction will be accepted, so processing of the instruction will be suspended. That is, execution of the instruction will continue on return from the interrupt processing routine. However, be sure to save the contents of the R1, R2, R3, and PSW when an interrupt is generated and restore them when execution is returned from the interrupt routine.

[Instruction Format]

Syntax	Processng Size	Code size (Byte)
SMOVF	В	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] SMOVF



SMOVU

Transferring a string

SMOVU

Strings MOVe while Unequal to zero

[Syntax] SMOVU

[Operation] unsigned char *R1, *R2, tmp; unsigned long R3; while (R3 != 0) { tmp = *R2++; *R1++ = tmp; R3--; if (tmp == '\0') { break; }

Note If this instruction is executed with R3 set to 0, it is ignored and has no effect on registers and flags. [Function]

This instruction transfers strings successively from the source address specified by R2 to the higher destination addresses specified by R1 until the null character "\0" (= 00h) is detected, with the number of bytes specified by R3 as the upper limit. String transfer is completed after the null character has been transferred.

In execution of the instruction, data may be prefetched from the source address specified by R2, with R3 as the upper limit. For details of the data size to be prefetched, refer to the hardware manual of each product.

The destination address specified by R1 should not be included in the range of data to be prefetched, which starts from the source address specified by R2.

The contents of R1 and R2 are undefined upon completion of the instruction.

An interrupt request during execution of this instruction will be accepted, so processing of the instruction will be suspended. That is, execution of the instruction will continue on return from the interrupt processing routine. However, be sure to save the contents of the R1, R2, R3, and PSW when an interrupt is generated and restore them when execution is returned from the interrupt routine.

[Instruction Format]

Syntax	Processng Size	Code size (Byte)
SMOVU	В	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] SMOVU

R20UT2999EJ0100 Rev.1.00 May. 20, 2014



SSTR

Storing a string

SSTR

String SToRe

[Syntax] SSTR.size

[Operation] unsigned { char | short | long } *R1, R2; unsigned long R3; while (R3 != 0) { *R1++ = R2; R3 = R3 - 1; } Note 1 If this instruction is even

- Note 1. If this instruction is executed with R3 set to 0, it is ignored and has no effect on registers and flags.
- Note 2. R1++: Incrementation is by the value corresponding to the size specifier (.size), i.e. by 1 for .B, 2 for .W, and 4 for .L.
- Note 3. R2: How much of the value in R2 is stored depends on the size specifier (.size): the byte from the LSB end of R2 is stored for .B, the word from the LSB end of R2 is stored for .W, and the longword in R2 is stored for .L.

[Function]

- This instruction stores the contents of R2 successively proceeding in the direction of increasing addresses specified by R1 up to the number specified by R3.
- On completion of instruction execution, R1 indicates the next address in sequence from that for the last transfer.
- An interrupt request during execution of this instruction will be accepted, so processing of the instruction will be suspended. That is, execution of the instruction will continue on return from the interrupt processing routine. However, be sure to save the contents of the R1, R2, R3, and PSW when an interrupt is generated and restore them when execution is returned from the interrupt routine.

[Instruction Format]

Syntax	size	Processng Size	Code size (Byte)
SSTR.size	B/W/L	L	2

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] SSTR.W



STNZ

Transfer with condition

STNZ

STore on Not Zero

[Syntax] STNZ src, dest

[Operation] if (Z == 0) dest = src;

[Function]

- This instruction moves src to dest when the Z flag is 0. dest does not change when the Z flag is 1.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
STNZ src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] STNZ #1, R2



STZ

Transfer with condition STZ

STore on Zero

[Syntax] STZ src, dest

[Operation] if (Z == 1) dest = src; [Function]

- This instruction moves src to dest when the Z flag is 1. dest does not change when the Z flag is 0.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
STZ src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] STZ #1, R2

[Related Instructions] STZ, STNZ



SUB

Subtraction without borrow

SUB

SUBtract

[Syntax] (1)SUB src, dest (2)SUB src, src2, dest

[Operation] (1)dest = dest - src; (2)dest = src2 - src;

[Function]

- (1)This instruction subtracts src from dest and places the result in dest.

- (2)This instruction subtracts src from src2 and places the result in dest.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	dest	Code size (Byte)
(1)SUB src, dest	L	#UIMM:4	_	Rd	2
	L	Rs	-	Rd	2
	L	[Rs].memex	_	Rd	2 (memex == UB) 3 (memex != UB)
	L	dsp:8[Rs].memex(注)	_	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:16[Rs].memex(注)	_	Rd	4 (memex == UB) 5 (memex != UB)
(2)SUB src, src2, dest	L	Rs	Rs2	Rd	3

[Flag Change]

Flag		С	Z	S	0	
0	Change	0	0			

Conditions

C : The flag is set if an unsigned operation produces no overflow; otherwise it is cleared.

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S : The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.

O : The flag is set if a signed operation produces an overflow; otherwise it is cleared.

[Description Example]

- SUB
 #15, R2

 SUB
 R1, R2

 SUB
 [R1], R2

 SUB
 1[R1].B, R2
- SUB R1, R2, R3



SUNTIL

Searching for a string Search UNTIL equal string

SUNTIL

[Syntax] SUNTIL.size

```
[Operation]

unsigned { char | short | long } *R1;

unsigned long R2, R3, tmp;

while ( R3 != 0 ) {

    tmp = ( unsigned long ) *R1++;

    R3--;

    if ( tmp == R2 ) {

        break;

    }

}
```

Note 1. If this instruction is executed with R3 set to 0, it is ignored and has no effect on registers and flags.

Note

2. R1++: Incrementation is by the value corresponding to the size specifier (.size), i.e. by 1 for .B, 2 for .W, and 4 for .L.

[Function]

- This instruction searches a string for comparison from the first address specified by R1 for a match with the value specified in R2, with the search proceeding in the direction of increasing addresses and the number specified by R3 as the upper limit on the number of comparisons. When the size specifier (.size) is .B or .W, the byte or word data on the memory is compared with the value in R2 after being zero-extended to form a longword of data.
- In execution of the instruction, data may be prefetched from the destination address for comparison specified by R1, with R3 as the upper limit. For details of the data size to be prefetched, refer to the hardware manual of each product.
- Flags change according to the results of the operation "*R1 R2".
- The value in R1 upon completion of instruction execution indicates the next address where the data matched. Unless there was a match within the limit, the value in R1 is the next address in sequence from that for the last comparison.
- The value in R3 on completion of instruction execution is the initial value minus the number of comparisons.
- An interrupt request during execution of this instruction will be accepted, so processing of the instruction will be suspended. That is, execution of the instruction will continue on return from the interrupt processing routine. However, be sure to save the contents of the R1, R2, R3, and PSW when an interrupt is generated and restore them when execution is returned from the interrupt routine.

[Instruction Format]

Syntax	size	Processng Size	Code size (Byte)
SUNTIL.size	B/W/L	L	2

[Flag Change]

Flag	С	Z	S	0
Change 🔘	0		_	_

Conditions

C : The flag is set if a comparison operation as unsigned integers results in any value equal to or greater than 0; otherwise it is cleared.

Z : The flag is set if matched data is found; otherwise it is cleared.

[Description Example] SUNTIL.W



SWHILE

Searching for a string

SWHILE

Search WHILE unequal string

```
[Syntax]
SWHILE.size
```

```
[Operation]

unsigned { char | short | long } *R1;

unsigned long R2, R3, tmp;

while ( R3 != 0 ) {

tmp = ( unsigned long ) *R1++;

R3--;

if ( tmp != R2 ) {

break;

}

}
```

Note

te 1. If this instruction is executed with R3 set to 0, it is ignored and has no effect on registers and flags.

Note

2. R1++: Incrementation is by the value corresponding to the size specifier (.size), i.e. by 1 for .B, 2 for .W, and 4 for .L.

[Function]

- This instruction searches a string for comparison from the first address specified by R1 for an unmatch with the value specified in R2, with the search proceeding in the direction of increasing addresses and the number specified by R3 as the upper limit on the number of comparisons. When the size specifier (.size) is. B or .W, the byte or word data on the memory is compared with the value in R2 after being zero-extended to form a longword of data.
- In execution of the instruction, data may be prefetched from the destination address for comparison specified by R1, with R3 as the upper limit. For details of the data size to be prefetched, refer to the hardware manual of each product.
- Flags change according to the results of the operation "*R1 R2".
- The value in R1 upon completion of instruction execution indicates the next addresses where the data did not match. If all the data contents match, the value in R1 is the next address in sequence from that for the last comparison.
- The value in R3 on completion of instruction execution is the initial value minus the number of comparisons.
- An interrupt request during execution of this instruction will be accepted, so processing of the instruction will be suspended. That is, execution of the instruction will continue on return from the interrupt processing routine. However, be sure to save the contents of the R1, R2, R3, and PSW when an interrupt is generated and restore them when execution is returned from the interrupt routine.

[Instruction Format]

Syntax	size	Processng Size	Code size (Byte)
SWHILE.size	B/W/L	L	2

[Flag Change]

Flag	С	Z	S	0
Change 🔿	0		_	_

Conditions

C : The flag is set if a comparison operation as unsigned integers results in any value equal to or greater than 0; otherwise it is cleared.

Z : The flag is set if all the data contents match; otherwise it is cleared.

[Description Example] SWHILE.W



TST

Logical test

TST

TeST logical

[Syntax] TST src, src2

[Operation] src2 & src; [Function]

- This instruction changes the flag states in the PSW according to the result of logical AND of src2 and src.

[Instruction Format]

Syntax	Pro- cessng Size	src	src2	Code size (Byte)
TST src, src2	L	#SIMM:8	Rs	4
	L	#SIMM:16	Rs	5
	L	#SIMM:24	Rs	6
	L	#IMM:32	Rs	7
	L	Rs	Rs2	3
		[Rs].memex	Rs2	3 (memex == UB) 4 (memex != UB)
		dsp:8[Rs].memex(注)	Rs2	4 (memex == UB) 5 (memex != UB)
		dsp:16[Rs].memex(注)	Rs2	5 (memex == UB) 6 (memex != UB)

[Flag Change]

Flag	С	Z	S	0
Char@e 🔿	_			_

Conditions

Z : The flag is set if the result of the operation is 0; otherwise it is cleared.

O : The flag is set if the MSB of the result of the operation is 1; otherwise it is cleared.

[Description Example]

TST #7, R2

TST R1, R2

TST [R1], R2

TST 1[R1].UB, R2



WAIT

Waiting

WAIT

WAIT

[Syntax] WAIT

[Operation]

[Function]

- This instruction stops program execution. Program execution is then restarted by acceptance of a non-maskable interrupt, interrupt, or generation of a reset.
- This instruction is a privileged instruction. Attempting to execute this instruction in user mode generates a privileged instruction exception.
- The I bit in the PSW becomes 1.
- The address of the PC saved at the generation of an interrupt is the one next to the WAIT instruction.
- Note For the power-down state when the execution of the program is stopped, refer to the hardware manual of each product.

[Instruction Format]

Syntax	Code size (Byte)
WAIT	2

[Flag Change]

- This instruction does not affect the states of flags. [Description Example] WAIT



XCHG

Exchanging values

XCHG

eXCHanGe

[Syntax] XCHG src, dest

[Operation] tmp = src; src = dest; dest = tmp;

[Function]

- This instruction exchanges the contents of src and dest as listed in the following table.

[Instruction Format]

Syntax	Pro- cessng Size	src	dest	Code size (Byte)
XCHG src, dest	L	Rs	Rd	3
	L	[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:8[Rs].memex(注)	Rd	4 (memex == UB) 5 (memex != UB)
	L	dsp:16[Rs].memex(注)	Rd	5 (memex == UB) 6 (memex != UB)

[Flag Change]

- This instruction does not affect the states of flags.

[Description Example] XCHG R1, R2 XCHG [R1].W, R2



XOR

Logical exclusive or

XOR

eXclusive OR logical

[Syntax] XOR src, dest

[Operation] dest = dest ^ src;

[Function]

- This instruction exclusive-ORs dest and src and places the result in dest.

[Instruction Format]

	Syntax	Pro- cessng Size	src	dest	Code size (Byte)
XOR	src, dest	L	#SIMM:8	Rd	4
		L	#SIMM:16	Rd	5
		L	#SIMM:24	Rd	6
		L	#IMM:32	Rd	7
		L	Rs	Rd	4
			[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
			dsp:8[Rs].memex(注)	Rd	4 (memex == UB) 5 (memex != UB)
			dsp:16[Rs].memex(注)	Rd	5 (memex == UB) 6 (memex != UB)

[Flag Change]

Flag	С	Z	S	0
Char@e 🔿	-			-

Conditions

Z : The flag is set if dest is 0 after the operation; otherwise it is cleared.

S : The flag is set if the MSB of dest after the operation is 1; otherwise it is cleared.

[Description Example]

XOR #8, R1 XOR R1, R2 XOR [R1], R2 XOR 16[R1].L, R2



5. LINK DIRECTIVE SPECIFICATIONS

This chapter explains the necessary items for link directives and how to write a directive file.

5.1 Section mapping

For section mapping, refer to RX Build / B.1.3 Options / (3) Optimizing Linkage Editor (rlink) Options / Section Options / -start.

5.2 Section type

For section types, refer to 3.5 List of Section Names.



6. Function Specifications

This chapter provides library functions supplied with the CCRX.

6.1 Supplied Libraries

The CCRX provides the standard C, standard C99, and EC++ libraries.

6.1.1 Terms Used in Library Function Descriptions

(1) Stream input/output

In data input/output, it would lead to poor efficiency if each call of an input/output function, which handles a single character, drove the input/output device and the OS functions. To solve this problem, a storage area called a buffer is normally provided, and the data in the buffer is input or output at one time.

From the viewpoint of the program, on the other hand, it is more convenient to call input/output functions for each character.

Using the library functions, character-by-character input/output can be performed efficiently without awareness of the buffer status within the program by automatically performing buffer management.

Those library functions enable a programmer to write a program considering the input/output as a single data stream, making the programmer be able to implement data input/output efficiently without being aware of the detailed procedure. Such capability is called stream input/output.

(2) FILE structure and file pointer

The buffer and other information required for the stream input/output described above are stored in a single structure, defined by the name **FILE** in the **<stdio.h>** standard include file.

In stream input/output, all files are handled as having a **FILE** structure data structure. Files of this kind are called stream files. A pointer to this **FILE** structure is called a file pointer, and is used to specify an input/output file. The file pointer is defined as

FILE *fp;

When a file is opened by the **fopen** function, etc., the file pointer is returned. If the open processing fails, **NULL** is returned. Note that if a **NULL** pointer is specified in another stream input/output function, that function will end abnormally. After opening a file, be sure to check the file pointer value to see whether the open processing has been successful.

(3) Functions and macros

There are two library function implementation methods: functions and macros. A function has the same interface as an ordinary user-written function, and is incorporated during linkage. A macro is defined using a **#define** statement in the standard include file relating to the function. The following points must be noted concerning macros:

- Macros are expanded automatically by the preprocessor, and therefore a macro expansion cannot be invalidated even if the user declares a function with the same name.
- If an expression with a side effect (assignment expression, increment, decrement) is specified as a macro parameter, its result is not guaranteed.

Example Macro definition of **MACRO** that calculates the absolute value of a parameter is as follows: If the following definition is made:

```
#define MACRO(a) ((a) >= 0 ? (a) : -(a))
and if
X=MACRO(a++)
is in the program, the macro will be expanded as follows:
X = ((a++) >= 0 ? (a++) : -(a++))
a will be incremented twice, and the resultant value will be different from the absolute value of the initial value of
a.
```

(4) **EOF**

In functions such as **getc**, **getchar**, and **fgetc**, which input data from a file, **EOF** is the value returned at end-offile. The name **EOF** is defined in the **<stdio.h>** standard include file.

(5) **NULL**

This is the value indicating that a pointer is not pointing at anything. The name **NULL** is defined in the **<std-def.h>** standard include file.



(6) Null character

The end of a string in C/C++ is indicated by the characters \0. String parameters in library functions must also conform to this convention. The characters \0 indicating the end of a string are called null characters.

(7) Return code

With some library functions, a return value is used to determine the result (such as whether the specified processing succeeded or failed). In this case, the return value is called the return code.

(8) Text files and binary files

Many systems have special file formats to store data. To support this facility, library functions have two file formats: text files and binary files.

- Text files

A text file is used to store ordinary text, and consists of a collection of lines. In text file input, the new-line character (\n) is input as a line separator. In output, output of the current line is terminated by outputting the newline character (\n). Text files are used to input/output files that store standard text for each system. With text files, characters input or output by a library function do not necessarily correspond to a physical stream of data in the file.

- Binary files

A binary file is configured as a row of byte data. Data input or output by a library function corresponds to a physical list of data in the file.

(9) Standard input/output files

Files that can be used as standard by input/output library functions by default without preparations such as opening file are called standard input/output files. Standard input/output files comprise the standard input file (**stdin**), standard output file (**stdout**), and standard error output file (**stderr**).

- Standard input file (stdin) Standard file to be input to a program.
- Standard output file (**stdout**) Standard file to be output from a program.
- Standard error output file (**stderr**) Standard file for storing output of error messages, etc., from a program.
- (10) Floating-point numbers

Floating-point numbers are numbers represented by approximation of real numbers. In a C source program, floating-point numbers are represented by decimal numbers, but inside the computer they are normally represented by binary numbers.

In the case of binary numbers, the floating-point representation is as follows:

 $2^n \times m$ (n: integer, m: binary fraction)

Here, **n** is called the exponent of the floating-point number, and **m** is called the mantissa. The numbers of bits to represent **n** and **m** are normally fixed so that a floating-point number can be represented using a specific data size.

Some terms relating to floating-point numbers are explained below.

- Radix

An integer value indicating the number of distinct digits in the number system used by a floating-point number (10 for decimal, 2 for binary, etc.). The radix is normally 2.

- Rounding

Rounding is performed when an intermediate result of an operation of higher precision than a floating-point type is stored as that floating-point type. There is rounding up, rounding down, and half-adjust rounding (i.e., in binary representation, rounding down 0 and rounding up 1).

- Normalization

When a floating-point number is represented in the form $2^n \times m$, the same number can be represented in different ways.

[Format] The following two expressions represent the same value.

 $2^{5} \times 1.0_{(2)}(2)$ indicates a binary number)

 $2^6 \times 0.1$ (2)

Usually, a representation in which the leading digit is not 0 is used, in order to secure the number of valid digits. This is called a normalized floating-point number, and the operation that converts a floating-point number to this kind of representation is called normalization.

- Guard bit

When saving an intermediate result of a floating-point operation, data one bit longer than the actual floatingpoint number is normally provided in order for rounding to be carried out. However, this alone does not permit



an accurate result to be achieved in the event of digit dropping, etc. For this reason, the intermediate result is saved with an extra bit, called a guard bit.

(11) File access mode

This is a string that indicates the kind of processing to be carried out on a file when it is opened. There are 12 different modes, as shown in table 6.1.

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

Table 6.1	File Access	Modes
		wouc

(12) Implementation definition Definitions differ for each compiler.

(13) Error indicator and end-of-file indicator

The following two data items are held for each stream file: (1) an error indicator that indicates whether or not an error has occurred during file input/output, and (2) an end-of-file indicator that indicates whether or not the input file has ended.

These data items can be referenced by the **ferror** function and the **feof** function, respectively. With some functions that handle stream files, error occurrence and end-of-file information cannot be obtained from the return value alone. The error indicator and end-of-file indicator are useful for checking the file status after execution of such functions.

(14) File position indicator

Stream files that can be read or written at any position within the file, such as disk files, have an associated data item called a file position indicator that indicates the current read/write position within the file. File position indicators are not used with stream files that do not permit the read/write position within the file to be changed, such as terminals.

6.1.2 Notes on Use of Libraries

The contents of macros defined in a library differ for each compiler. When a library is used, the behavior is not guaranteed if the contents of these macros are redefined. With libraries, errors are not detected in all cases. The behavior is not guaranteed if library functions are called in a form other than those shown in the descriptions in the following sections.

6.2 Header Files

The list of header files required for using the libraries of the RX are listed below. The macro definitions and function declarations are described in each file.



Library Type	Description	Standard Include File
Program diagnostics	Outputs program diagnostic information.	<assert.h></assert.h>
Character handling	Handles and checks characters.	<ctype.h></ctype.h>
Mathematics	Performs numerical calculations such as trigonomet- ric functions.	<math.h> <mathf.h></mathf.h></math.h>
Non-local jumps	Supports transfer of control between functions.	<setjmp.h></setjmp.h>
Variable arguments	Supports access to variable arguments for functions with such arguments.	<stdarg.h></stdarg.h>
Input/output	Performs input/output handling.	<stdio.h></stdio.h>
General utilities	Performs C program standard processing such as storage area management.	<stdlib.h></stdlib.h>
String handling	Performs string comparison, copying, etc.	<string.h></string.h>
Complex arithmetic	Performs complex number operations.	<complex.h></complex.h>
Floating-point environment	Supports access to floating-point environment.	<fenv.h></fenv.h>
Integer type format conver- sion	Manipulates greatest-width integers and converts integer format.	<inttypes.h></inttypes.h>
Multibyte and wide charac- ters	Manipulates multibyte characters.	<wchar.h> <wctype.h></wctype.h></wchar.h>

Table 0.2 Library Types and Concepting Standard Include The	Table 6.2	Library Types and Corresponding Standard Include Files
---	-----------	--

In addition to the above standard include files, standard include files consisting solely of macro name definitions, shown in table 6.3, are provided to improve programming efficiency.

Table 6.3 Standard Include Files Comprising Macro Name Definitions

Standard Include File	Description
<stddef.h></stddef.h>	Defines macro names used in common by the standard include files.
<limits.h></limits.h>	Defines various limit values relating to compiler internal processing.
<errno.h></errno.h>	Defines the value to be set in errno when an error is generated in a library function.
<float.h></float.h>	Defines various limit values relating to the limits of floating-point numbers.
<iso646.h></iso646.h>	Defines alternative spellings of macro names.
<stdbool.h></stdbool.h>	Defines macros relating to logical types and values.
<stdint.h></stdint.h>	Declares integer types with specified width and defines macros.
<tgmath.h></tgmath.h>	Defines type-generic macros.

6.3 Reentrant Library

Table 6.4 lists libraries that are reentrant. 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**.

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
stddef.h	offsetof	0	math.h	frexp	D
assert.h	assert	Х		ldexp	D
ctype.h	isalnum	0		log	D
	isalpha	0		log10	D
	iscntrl	0		modf	D
	isdigit	0		pow	D
	isgraph	0		sqrt	D ^{*1}
	islower	0		ceil	D
	isprint	0		fabs	D
	ispunct	0		floor	D
	isspace	0		fmod	D
	isupper	0	mathf.h	acosf	D
	isxdigit	0		asinf	D
	tolower	0		atanf	D
	toupper	0		atan2f	D
math.h	acos	D		cosf	D
	asin	D		sinf	D
	atan	D		tanf	D
	atan2	D		coshf	D
	cos	D		sinhf	D
	sin	D		tanhf	D
	tan	D		expf	D
	cosh	D		frexpf	D
	sinh	D		ldexpf	D
	tanh	D		logf	D
	ехр	D		log10f	D

Table 6.4 Reentrant Library List



Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
mathf.h	modff	D	stdio.h	fputs	Х
	powf	D		getc	Х
	sqrtf	D *1		getchar	Х
	ceilf	D		gets	Х
	fabsf	D		putc	Х
	floorf	D		putchar	Х
	fmodf	D		puts	Х
setjmp.h	setjmp	0		ungetc	Х
	longjmp	0		fread	Х
stdarg.h	va_start	0		fwrite	Х
	va_arg	0		fseek	Х
	va_end	0		ftell	Х
stdio.h	fclose	Х		rewind	Х
	fflush	Х		clearerr	Х
	fopen	Х		feof	Х
	freopen	Х		ferror	Х
	setbuf	Х		perror	Х
	setvbuf	Х	stdlib.h	atof	D
	fprintf	Х		atoi	D
	fscanf	Х		atol	D
	printf	Х		atoll	D
	scanf	Х		strtod	D
	sprintf	D		strtol	D
	sscanf	D		strtoul	D
	vfprintf	X		strtoll	D
	vprintf	X		strtoull	D
	vsprintf	D		rand	Х
	fgetc	X		srand	Х
	fgets	X		calloc	Х
	fputc	Х		free	Х

Standard Include File	Function Name	Reentrant	Standard Include File	Function Name	Reentrant
stdlib.h	malloc	Х	string.h	тетстр	0
	realloc	х		strcmp	0
	bsearch	0		strncmp	0
	qsort	0		memchr	0
	abs	0		strchr	0
	div	D		strcspn	0
string.h	labs	0		strpbrk	0
	llabs	0		strrchr	0
	ldiv	D		strspn	0
	lldiv	D		strstr	0
	тетсру	0		strtok	Х
	strcpy	0		memset	0
	strncpy	0		strerror	0
	strcat	0		strlen	0
	strncat	0		memmove	0

Reentrant column:

O: Reentrant

X: Non-reentrant

 $\Delta : \textbf{errno} \text{ is set.}$

Notes 1.

This becomes reentrant if the function sqrtf is replaced by the FSQRT instruction, since the errno variable will not be updated. Regarding the conditions for replacement by the FSQRT instruction, refer to the corresponding item in the -library compiler options.



6.4 Library Function

This section explains library functions.

Some of the C99-language-expanded keywords (functions, macros, variable names, etc..) must be used in when the C99-language is selected. Such keywords are displayed by the mark of "<-lang=c99>" at the tables each the header-files in these sections. When you use these keywords in your program, at the time of compilations and library generations, please turn on the -lang=c99 option.

6.4.1 <stddef.h>

Defines macro names used in common in the standard include files. The following macro names are all implementation-defined.

Туре	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 possi- ble to check whether an error occurred during the library function process- ing.
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

6.4.2 <assert.h>

Adds diagnostics into programs.

Туре	Definition Name	Description
Function (macro)	assert	Adds diagnostics into programs.

To invalidate the diagnostics defined by **<assert.h>**, define macro name **NDEBUG** with a **#define** statement (**#define NDEBUG**) before including **<assert.h>**.

Note If an **#undef** statement is used for macro name **assert**, the result of subsequent **assert** calls is not guaranteed.

assert

Adds diagnostics into programs. [Format]



#include <assert.h>
void assert (long expression)
[Parameters]
expressionExpression to be evaluated.
[Remarks]

When **expression** is true, the **assert** macro terminates processing without returning a value. If **expression** is false, it outputs diagnostic information to the standard error file in the form defined by the compiler, and then calls the **abort** function.

The diagnostic information includes the parameter's program text, source file name, and source line numbers. Implementation define:

The following message is output when **expression** is false in **assert (expression)**: The message depends on the **lang** option setting at compilation.

(1) When -lang=c99 is not specified (C (C89), C++, or EC++ language):

ASSERTION FAILED:expressionFILE<file name>,

LINE<line number>

(2) When **-lang=c99** is specified (C (C99) language):

ASSERTION FAILED:expressionFILE<file name>, LINE<line number>FUNCNAME<function name>

6.4.3 <ctype.h>

Checks and converts character types.

Туре	Definition Name	Description
Function	isalnum	Tests for a letter or a decimal digit.
	isalpha	Tests for a letter.
	iscntrl	Tests for a control character.
	isdigit	Tests for a decimal digit.
	isgraph	Tests for a printing character except space.
	islower	Tests for a lowercase letter.
	isprint	Tests for a printing character including space.
	ispunct	Tests for a special character.
	isspace	Tests for a white-space character.
	isupper	Tests for an uppercase letter.
	isxdigit	Tests for a hexadecimal digit.
	tolower	Converts an uppercase letter to lowercase.
	toupper	Converts a lowercase letter to uppercase.
	isblank <-lang=c99>	Tests for a space character or a tab character.

In the above functions, if the input parameter value is not within the range that can be represented by the **unsigned char** type and is not **EOF**, the operation of the function is not guaranteed.

Character types are listed in table 6.5.

Table 6.5	Character Types	5
-----------	-----------------	---

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'

Character Type	Description
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 6.6 shows the character set that results in a true return value.

Table 6.6 True Character

Function Name	True Characters	
isalnum	'0' to '9', 'A' to 'Z', 'a' to 'z'	
isalpha	'A' to 'Z', 'a' to 'z'	
iscntrl	'\x00' to '\x1f', '\x7f'	
islower	'a' to 'z'	
isprint	'\x20' to '\x7E'	
isupper	'A' to 'Z'	

isalnum

Tests for a letter or a decimal digit. [Format] #include <ctype.h> long isalnum (long c); [Parameters] c Character to be tested [Return values] If character **c** is a letter or a decimal digit: Nonzero If character **c** is not a letter or a decimal digit: 0

isalpha



Tests for a letter. [Format] #include <ctype.h> long isalpha(long c); [Parameters] c Character to be tested [Return values] If character **c** is a letter: Nonzero If character **c** is not a letter: 0

iscntrl

Tests for a control character. [Format] #include <ctype.h> long iscntrl (long c); [Parameters] c Character to be tested [Return values] If character **c** is a control character: Nonzero If character **c** is not a control character: 0

isdigit

Tests for a decimal digit. [Format] #include <ctype.h> long isdigit (long c); [Parameters] c Character to be tested [Return values] If character c is a decimal digit: Nonzero If character c is not a decimal digit: 0

isgraph

Tests for any printing character except space (' '). [Format] #include <ctype.h> long isgraph (long c); [Parameters] c Character to be tested [Return values] If character c is a printing character except space: Nonzero If character c is not a printing character except space: 0

islower

Tests for a lowercase letter. [Format] #include <ctype.h> long islower (long c); [Parameters] c Character to be tested [Return values] If character c is a lowercase letter: Nonzero If character c is not a lowercase letter: 0

isprint

Tests for a printing character including space (' ').



[Format]
#include <ctype.h>
long isprint (long c);
[Parameters]
c Character to be tested
[Return values]
If character c is a printing character including space: Nonzero
If character c is not a printing character including space: 0

ispunct

Tests for a special character. [Format] #include <ctype.h> long ispunct (long c); [Parameters] c Character to be tested [Return values] If character c is a special character: Nonzero If character c is not a special character: 0

isspace

Tests for a white-space character. [Format] #include <ctype.h> long isspace (long c); [Parameters] c Character to be tested [Return values] If character c is a white-space character: Nonzero If character c is not a white-space character: 0

isupper

Tests for an uppercase letter. [Format] #include <ctype.h> long isupper (long c); [Parameters] c Character to be tested [Return values] If character **c** is an uppercase letter: Nonzero If character **c** is not an uppercase letter: 0

isxdigit

Tests for a hexadecimal digit. [Format] #include <ctype.h> long isxdigit (long c); [Parameters] c Character to be tested [Return values] If character **c** is a hexadecimal digit: Nonzero If character **c** is not a hexadecimal digit: 0

tolower



Converts an uppercase letter to the corresponding lowercase letter. [Format] #include <ctype.h> long tolower (long c); [Parameters] c Character to be converted [Return values] If character c is an uppercase letter:Lowercase letter corresponding to character c If character c is not an uppercase letter: Character c

toupper

Converts a lowercase letter to the corresponding uppercase letter. [Format] #include <ctype.h> long toupper (long c); [Parameters] c Character to be converted [Return values] If character **c** is a lowercase letter: Uppercase letter corresponding to character **c** If character **c** is not a lowercase letter: Character **c**

isblank

Tests for a space character or a tab character. [Format] #include <ctype.h> long isblank (long c); [Parameters] c Character to be tested [Return values] If character c is a space character or a tab character: Nonzero If character c is neither a space character nor a tab character: 0

6.4.4 <float.h>

Defines various limits relating to the internal representation of floating-point numbers. The following macro names are all implementation-defined.

Туре	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 fol- lows: 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.



Туре	Definition Name	Definition Value	Description
Constant (macro)	FLT_MAX	3.4028235677973364e +38F	Indicates the maximum value that can be represented as a float type floating-point value.
	DBL_MAX	1.7976931348623158e +308	Indicates the maximum value that can be represented as a double type floating-point value.
	LDBL_MAX	1.7976931348623158e +308	Indicates the maximum value that can be repre- sented 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 float- ing-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 float-ing-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.
	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.


Туре	Definition Name	Definition Value	Description
Constant (macro)	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 dou- ble type floating-point value decimal- precision.
	LDBL_DIG	15	Indicates the maximum number of digits in long double type floating-point value decimal-preci- sion.
	FLT_MANT_DIG	24	Indicates the maximum number of mantissa dig- its when a float type floating-point value is rep- resented in the radix.
	DBL_MANT_DIG	53	Indicates the maximum number of mantissa dig- its when a double type floating-point value is represented in the radix.
	LDBL_MANT_DIG	53	Indicates the maximum number of mantissa dig- its when a long double type floating-point value is represented in the radix.
	FLT_EXP_DIG	8	Indicates the maximum number of exponent dig- its when a float type floating-point value is rep- resented in the radix.
	DBL_EXP_DIG	11	Indicates the maximum number of exponent dig- its when a double type floating-point value is represented in the radix.
	LDBL_EXP_DIG	11	Indicates the maximum number of exponent dig- its 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 $1.0 + x \neq 1.0$ in float type.
	DBL_POS_EPS	1.1102230246251567e-16	Indicates the minimum floating-point value x for which $1.0 + x \neq 1.0$ in double type.
	LDBL_POS_EPS	1.1102230246251567e-16	Indicates the minimum floating-point value x for which $1.0 + x \neq 1.0$ in long double type.
	FLT_NEG_EPS	2.9802324164052156e-8F	Indicates the minimum floating-point value x for which $1.0 - x \neq 1.0$ in float type.
	DBL_NEG_EPS	5.5511151231257834e-17	Indicates the minimum floating-point value x for which $1.0 - x \neq 1.0$ in double type
	LDBL_NEG_EPS	5.5511151231257834e-17	Indicates the minimum floating-point value x for which $1.0 - x \neq 1.0$ in long double type.



Туре	Definition Name	Definition Value	Description	
Constant (macro)	FLT_POS_EPS_ EXP	-23	Indicates the minimum integer n for which 1.0 + $(radix)^n \neq 1.0$ in float type.	
	DBL_POS_EPS_ EXP	-52	Indicates the minimum integer n for which 1.0 $+(radix)^n \neq 1.0$ in double type.	
	LDBL_POS_EPS_EX P	-52	Indicates the minimum integer n for which 1.0 + $(radix)^n \neq 1.0$ in long double type.	
	FLT_NEG_EPS_ EXP	-24	Indicates the minimum integer n for which $1.0 - (radix)^n \neq 1.0$ in float type.	
	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_EX P	-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 mini- mum value greater than 1 that can be repre- sented in float type.	
	DBL_EPSILON	1E-9	Indicates the difference between 1 and the mini- mum value greater than 1 that can be repre- sented in double type.	
	LDBL_EPSILON	1E–9	Indicates the difference between 1 and the mini- mum value greater than 1 that can be repre- sented in long double type.	

6.4.5 mits.h>

Defines various limits relating to the internal representation of integer type data. The following macro names are all implementation-defined.

Туре	Definition Name Definition Value		Description
Constant (macro)	CHAR_BIT	8	Indicates the number of bits in a char type value.
	CHAR_MAX	127	Indicates the maximum value that can be repre-
		255* ¹	sented by a char type variable.
	CHAR_MIN	-128	Indicates the minimum value that can be repre-
		0*1	sented 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 repre- sented by an unsigned char type variable.



Туре	Definition Name	Definition Value	Description
Constant (macro)	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	Indicates the maximum value that can be represented by an int type variable.
	INT_MIN	-2147483647-1	Indicates the minimum value that can be represented by an int type variable.
	UINT_MAX	4294967295U	Indicates the maximum value that can be represented by an unsigned int type variable.
	LONG_MAX	217483647L	Indicates the maximum value that can be represented by a long type variable.
	LONG_MIN	-2147483647L-1L	Indicates the minimum value that can be represented by a long type variable.
	ULONG_MAX	4294967295U	Indicates the maximum value that can be represented by an unsigned long type variable.
	LLONG_MAX	9223372036854775807LL	Indicates the maximum value that can be represented by a long long type variable.
	LLONG_MIN	-9223372036854775807LL -1LL	Indicates the minimum value that can be represented by a long long type variable.
	ULLONG_MAX	18446744073709551615ULL	Indicates the maximum value that can be represented by an unsigned long long type variable.

Notes 1. Indicates the value that can be represented by a variable when the **signed_char** option is specified.

6.4.6 <errno.h>

Defines the value to be set in **errno** when an error is generated in a library function. The following macro names are all implementation-defined.

Туре	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			



6.4.7 <math.h>

Performs various mathematical operations.

The following constants (macros) are all implementation-defined.

Туре	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 can- not be represented as a double type value, or if an overflow or an underflow occurs.
	HUGE_VAL HUGE_VALF <-lang=c99> HUGE_VALL <-lang=c99>	Indicates the value for the function return value if the result of a function overflows.
	INFINITY <-lang=c99>	Expanded to a float -type constant expression that represents positive or unsigned infinity.
	NAN <-lang=c99>	Defined when float -type qNaN is supported.
	FP_INFINITE<-lang=c99>FP_NAN<-lang=c99>FP_NORMAL<-lang=c99>FP_SUBNORMAL<-lang=c99>FP_ZERO<-lang=c99>	These indicate exclusive types of floating-point values.
	FP_FAST_FMA<-lang=c99>FP_FAST_FMA<-lang=c99>FFP_FAST_FMAFL <-lang=c99>	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<-lang=c99>FP_ILOGBNAN<-lang=c99>	These are expanded to an integer constant expression of the value returned by ilogb when they are 0 or not-a-number, respectively.
	MATH_ERRNO <-lang=c99> MATH_ERREXCEPT <-lang=c99>	These are expanded to integer constants 1 and 2, respectively.
	math_errhandling <-lang=c99>	Expanded to an int -type expression whose value is a bitwise logical OR of MATH_ERRNO and MATH_ERREXCEPT .
Туре	float_t <-lang=c99> double_t <-lang=c99>	These are floating-point types having the same width as float and double , respectively.



Туре	Definition Name	Description
Function (macro)	fpclassify <-lang=c99>	Classifies argument values into not-a-number, infinity, normalized number, denormalized number, and 0.
	isfinite <-lang=c99>	Determines whether the argument is a finite value.
	isinf <-lang=c99>	Determines whether the argument is infinity.
	isnan <-lang=c99>	Determines whether the argument is a not-a-number.
	isnormal <-lang=c99>	Determines whether the argument is a normalized number.
	signbit <-lang=c99>	Determines whether the sign of the argument is negative.
	isgreater <-lang=c99>	Determines whether the first argument is greater than the second argument.
	isgreaterequal <-lang=c99>	Determines whether the first argument is equal to or greater than the second argument.
	isless <-lang=c99>	Determines whether the first argument is smaller than the second argument.
	islessequal <-lang=c99>	Determines whether the first argument is equal to or smaller than the second argument.
	islessgreater <-lang=c99>	Determines whether the first argument is smaller or greater than the second argument.
	isunordered <-lang=c99>	Determines whether the arguments are not ordered.
Function	acos acosf acosl	Calculates the arc cosine of a floating-point number.
	asin asinf asinl	Calculates the arc sine of a floating-point number.
	atan atanf atanl	Calculates the arc tangent of a floating-point number.
	atan2 atan2f atan2l	Calculates the arc tangent of the result of a division of two floating-point numbers.
	cos cosf cosl	Calculates the cosine of a floating-point radian value.

Туре	Definition Name	Description
Function	sin sinf sinl	Calculates the sine of a floating-point radian value.
	tan tanf tanl	Calculates the tangent of a floating-point radian value.
	cosh coshf coshl	Calculates the hyperbolic cosine of a floating-point number.



Туре	Definition Name	Description
	sinh sinhf sinhl	Calculates the hyperbolic sine of a floating-point number.
	tanh tanhf tanhl	Calculates the hyperbolic tangent of a floating-point number.
	exp expf expl	Calculates the exponential function of a floating-point number.
	frexp frexpf frexpl	Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.
	ldexp ldexpf ldexpl	Multiplies a floating-point number by a power of 2.
	log logf logl	Calculates the natural logarithm of a floating-point number.
	log10 log10f log10l	Calculates the base-ten logarithm of a floating-point number.
	modf modff modfl	Breaks a floating-point number into integral and fractional parts.
	pow powf powl	Calculates a power of a floating-point number.
	sqrt sqrtf sqrtl	Calculates the positive square root of a floating-point number.
	ceil ceilf ceill	Calculates the smallest integral value not less than or equal to the given floating-point number.
Function	fabs fabsf fabsl	Calculates the absolute value of a floating-point number.
	floor floorf floorl	Calculates the largest integral value not greater than or equal to the given floating-point number.
	fmod fmodf fmodl	Calculates the remainder of a division of two floating-point numbers.
	acosh <-lang=c99> acoshf <-lang=c99> acoshl <-lang=c99>	Calculates the hyperbolic arc cosine of a floating-point number.
	asinh<-lang=c99>asinhf<-lang=c99>asinhl<-lang=c99>	Calculates the hyperbolic arc sine of a floating-point number.



Туре	Defi	nition Name	Description
	atanh atanhf atanhl	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the hyperbolic arc tangent of a floating-point number.
	exp2 exp2f exp2l	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the value of 2 raised to the power x .
	expm1 expm1f expm1l	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the natural logarithm raised to the power x and subtracts 1 from the result.
	ilogb ilogbf ilogbl	<-lang=c99> <-lang=c99> <-lang=c99>	Extracts the exponent of x as a signed int value.
	log1p log1pf log1pl	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the natural logarithm of the argument + 1.
	log2 log2f log2l	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the base-2 logarithm.
	logb logbf logbl	<-lang=c99> <-lang=c99> <-lang=c99>	Extracts the exponent of x as a signed integer.
	scalbn scalbnf scalbnl scalbln scalblnf scalblnl	<-lang=c99> <-lang=c99> <-lang=c99> <-lang=c99> <-lang=c99> <-lang=c99>	Calculates x × FLT_RADIXn .
	cbrt cbrtf cbrtl	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the cube root of a floating-point number.
Function	hypot hypotf hypotl	<-lang=c99> <-lang=c99> <-lang=c99>	Raises each floating-point number to the power 2 and calculates the sum of the resultant values.
	erf erff erfl	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the error function.
	erfc erfcf erfcl	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the complementary error function.
	lgamma Igammaf Igammal	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the natural logarithm of the absolute value of the gamma function.
	tgamma tgammaf tgammal	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates the gamma function.
	nearbyint nearbyintf nearbyintl	<-lang=c99> <-lang=c99> <-lang=c99>	Rounds a floating-point number to an integer in the floating-point repre- sentation according to the current rounding direction.



Туре	Definition Name	Description
	rint <-lang=c99> rintf <-lang=c99> rintl <-lang=c99>	Equivalent to nearbyint except that this function group may generate floating-point exception.
	Irint<-lang=c99>Irintf<-lang=c99>Irintl<-lang=c99>Ilrint<-lang=c99>Ilrintf<-lang=c99>Ilrintl<-lang=c99>	Rounds a floating-point number to the nearest integer according to the rounding direction.
	round <-lang=c99> roundf <-lang=c99> roundl <-lang=c99>	Rounds a floating-point number to the nearest integer in the floating- point representation.
	Iround<-lang=c99>Iroundf<-lang=c99>Iroundl<-lang=c99>Ilround<-lang=c99>Ilroundf<-lang=c99>Ilroundl<-lang=c99>	Rounds a floating-point number to the nearest integer.
	trunc<-lang=c99>truncf<-lang=c99>truncl<-lang=c99>	Rounds a floating-point number to the nearest integer.
	remainder <-lang=c99> remainderf <-lang=c99> remainderl <-lang=c99>	Calculates remainder x REM y specified in the IEEE60559 standard.
	remquo <-lang=c99> remquof <-lang=c99> remquol <-lang=c99>	Calculates the value having the same sign as x/y and the absolute value congruent modulo- 2^n to the absolute value of the quotient.
Function	copysign<-lang=c99copysignf<-lang=c99	Generates a value consisting of the given absolute value and sign.
	nan <-lang=c99> nanf <-lang=c99> nanl <-lang=c99>	<pre>nan("n string") is equivalent to ("NAN(n string)", (char**) NULL).</pre>
	nextafter <-lang=c99> nextafterf <-lang=c99> nextafterl <-lang=c99>	Converts a floating-point number to the type of the function and calcu- lates the representable value following the converted number on the real axis.
	nexttoward <-lang=c99> nexttowardf <-lang=c99> nexttowardl <-lang=c99>	Equivalent to the nextafter function group except that the second argument is of type long double and returns the second argument after conversion to the type of the function.
	fdim <-lang=c99> fdimf <-lang=c99> fdiml <-lang=c99>	Calculates the positive difference.
	fmax<-lang=c99>fmaxf<-lang=c99>fmaxl<-lang=c99>	Obtains the greater of two values.
	fmin<-lang=c99>fminf<-lang=c99>fminl<-lang=c99>	Obtains the smaller of two values.



Туре	Definition Name		Description
	fma fmaf fmal	<-lang=c99> <-lang=c99> <-lang=c99>	Calculates (d1 * d2) + d3 as a single ternary operation.

Operation in the event of an error is described below.

(1) Domain error

A domain error occurs if the value of a parameter input to a function is outside the domain over which the mathematical function is defined. In this case, the value of **EDOM** is set in **errno**. The function return value in implementation-defined.

(2) Range error

A range error occurs if the result of a function cannot be represented as a value of the double type. In this case, the value of **ERANGE** is set in **errno**. If the result overflows, the function returns the value of **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** with the same sign as the correct value of the function. If the result underflows, 0 is returned as the return value.

Notes 1. If there is a possibility of a domain error resulting from a **<math.h>** function call, it is dangerous to use the resultant value directly. The value of **errno** should always be checked before using the result in such cases.

[Format]

```
1 x=asin(a);
2 if (errno==EDOM)
3 printf ("error\n");
4 else
5 printf ("result is : %lf\n",x);
.
```

In line 1, the arc sine value is computed using the **asin** function. If the value of argument **a** is outside the **asin** function domain [-1.0, 1.0], the **EDOM** value is set in **errno**. Line 2 determines whether a domain error has occurred. If a domain error has occurred, **error** is output in line 3. If there is no domain error, the arc sine value is output in line 5.

Notes 2. Whether or not a range error occurs depends on the internal representation format of floating-point types determined by the compiler. For example, if an internal representation format that allows an infinity to be represented as a value is used, **<math.h>** library functions can be implemented without causing range errors.

Implementation-Defined Specifications

Item	Compiler Specifications
Value returned by a mathematical function if an input argument is out of the range	A not-a-number is returned. For details on the format of not-a-numbers, refer to section 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.

acos/acosf/acosl

Calculates the arc cosine of a floating-point number. [Format]



#include <math.h>
double acos (double d)
float acosf (float d)
long double acosl (long double d);
[Parameters]
d Floating-point number for which arc cosine is to be computed
[Return values]
Normal:Arc cosine of d
Abnormal: Domain error: Returns not-a-number.
[Remarks]
A domain error occurs for a value of d not in the range [-1.0, +1.0].
The acos function returns the arc cosine in the range [0, π] by the radian.

asin/asinf/asinl

```
Calculates the arc sine of a floating-point number.

[Format]

#include <math.h>

double asin (double d)

float asinf (float d)

long double asinl (long double);

[Parameters]

d Floating-point number for which arc sine is to be computed

[Return values]

Normal:Arc sine of d

Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs for a value of d not in the range [-1.0, +1.0].

The asin function returns the arc sine in the range [-π/2, +π/2] by the radian.
```

atan/atanf/atanl

Calculates the arc tangent of a floating-point number. [Format] #include <math.h> double atan (double d) float atanf (float d) long double atanl (long double d); [Parameters] d Floating-point number for which arc tangent is to be computed [Return values] Arc tangent of d [Remarks] The atan function returns the arc tangent in the range (-π/2, +π/2) by the radian.

atan2/atan2f/atan2l

Calculates the arc tangent of the division of two floating-point numbers. [Format] #include <math.h> double atan2 (double y, double x) float atan2f (float y, float x) long double atan2l (long double y, long double x); [Parameters] x Divisor y Dividend [Return values] Normal:Arc tangent value when y is divided by x Abnormal: Domain error: Returns not-a-number. [Remarks]



A domain error occurs if the values of both x and y are 0.0.

The atan2 function returns the arc tangent in the range $(-\pi, +\pi)$ by the radian. The meaning of the atan2 function is illustrated in figure 6.1. As shown in the figure, the result of the atan2 function is the angle between the X-axis and a straight line passing through the origin and point (x, y).

If y = 0.0 and x is negative, the result is π . If x = 0.0, the result is $\pm \pi/2$, depending on whether y is positive or negative.

Figure 6.1 Meaning of atan2 Function



cos/cosf/cosl

Calculates the cosine of a floating-point radian value. [Format] #include <math.h> double cos (double d) float cosf (float d) long double cosl (long double d); [Parameters] d Radian value for which cosine is to be computed [Return values] Cosine of d

sin/sinf/sinl

Calculates the sine of a floating-point radian value.
[Format]
#include <math.h>
double sin (double d)
float sinf (float d)
long double sinl (long double d);
[Parameters]
d Radian value for which sine is to be computed
[Return values]
Sine of d

tan/tanf/tanl

Calculates the tangent of a floating-point radian value. [Format] #include <math.h> double tan (double d) float tanf (float d) long double tanl (long double d); [Parameters] d Radian value for which tangent is to be computed [Return values] Tangent of d

cosh/coshf/coshl

Calculates the hyperbolic cosine of a floating-point number. [Format]



#include <math.h>
double cosh (double d)
float coshf (float d)
long double coshl (long double d);
[Parameters]
d Floating-point number for which hyperbolic cosine is to be computed
[Return values]
Hyperbolic cosine of d

sinh/sinhf/sinhl

Calculates the hyperbolic sine of a floating-point number. [Format] #include <math.h> double sinh (double d) float sinhf (float d) long double sinhl (long double d); [Parameters] d Floating-point number for which hyperbolic sine is to be computed [Return values] Hyperbolic sine of d

tanh/tanhf/tanhl

Calculates the hyperbolic tangent of a floating-point number.

[Format]
#include <math.h>
double tanh (double d)
float tanhf (float d)
long double tanhl (long double d);
[Parameters]
d Floating-point number for which hyperbolic tangent is to be computed
[Return values]
Hyperbolic tangent of d

exp/expf/expl

Calculates the exponential function of a floating-point number. [Format] #include <math.h> double exp (double d) float expf (float d) long double expl (long double d); [Parameters] d Floating-point number for which exponential function is to be computed [Return values] Exponential function value of d

frexp/frexpf/frexpl

Breaks a floating-point number into a [0.5, 1.0) value and a power of 2. [Format] #include <math.h> double frexp (double value, double long *exp); float frexpf (float value, long * exp); long double frexpl (long double value, long *exp); [Parameters] value Floating-point number to be broken into a [0.5, 1.0) value and a power of 2 exp Pointer to storage area that holds power-of-2 value



[Return values] If value is 0.0: 0.0 If value is not 0.0: Value of ret defined by ret * 2^{value pointed to by exp} = value [Remarks] The frexp function breaks value into a [0.5, 1.0) value and a power of 2. It stores the resultant power-of-2 value in the area pointed to by exp. The frexp function returns the return value ret in the range [0.5, 1.0) or as 0.0.

If value is 0.0, the contents of the int storage area pointed to by exp and the value of ret are both 0.0.

ldexp/ldexpf/ldexpl

Multiplies a floating-point number by a power of 2. [Format] #include <math.h> double ldexp (double e, long f); float ldexpf (float e, long f); long double ldexpl (long double e, long f); [Parameters] e Floating-point number to be multiplied by a power of 2 f Power-of-2 value [Return values] Result of e * 2^f operation

log/logf/logl

Calculates the natural logarithm of a floating-point number. [Format] #include <math.h> double log (double d); float logf (float d); long double logl (long double d); [Parameters] d Floating-point number for which natural logarithm is to be computed [Return values] Normal: Natural logarithm of d Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if d is negative. A range error occurs if d is 0.0.

log10/log10f/log10l

Calculates the base-ten logarithm of a floating-point number. [Format] #include <math.h> double log10 (double d); float log10f(float d); long double log10l(long double d); [Parameters] d Floating-point number for which base-ten logarithm is to be computed [Return values] Normal: Base-ten logarithm of d Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if d is negative. A range error occurs if d is 0.0.

modf/modff/modfl



Breaks a floating-point number into integral and fractional parts. [Format] #include <math.h> double modf (double a, double*b); float modff (float a, float *b); long double modfl (long double a, long double *b); [Parameters] a Floating-point number to be broken into integral and fractional parts b Pointer indicating storage area that stores integral part [Return values] Fractional part of a

pow/powf/powl

Calculates a power of floating-point number. [Format] #include <math.h> double pow (double x, double y); float powf (float x, float y); long double powl (long double x, long double y); [Parameters] x Value to be raised to a power y Power value [Return values] Normal: Value of x raised to the power y Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if x is 0.0 and y is 0.0 or less, or if x is negative and y is not an integer.

sqrt/sqrtf/sqrtl

Calculates the positive square root of a floating-point number. [Format] #include <math.h> double sqrt (double d); float sqrtf (float d); long double sqrtl (long double d); [Parameters] d Floating-point number for which positive square root is to be computed [Return values] Normal: Positive square root of d Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if d is negative.

ceil/ceilf/ceill

Returns the smallest integral value not less than or equal to the given floating-point number.

[Format]
#include <math.h>
double ceil (double d);
float ceilf (float d);
long double ceill (long double d);
[Parameters]
d Floating-point number for which smallest integral value not less than that number is to be computed
[Return values]
Smallest integral value not less than or equal to d
[Remarks]
The ceil function returns the smallest integral value not less than or equal to d, expressed as a double ty

The ceil function returns the smallest integral value not less than or equal to d, expressed as a double type value. Therefore, if d is negative, the value after truncation of the fractional part is returned.



fabs/fabsf/fabsl

```
Calculates the absolute value of a floating-point number.
[Format]
#include <math.h>
double fabs (double d);
float fabsf (float d);
long double fabsl (long double d);
[Parameters]
d Floating-point number for which absolute value is to be computed
[Return values]
Absolute value of d
```

floor/floorf/floorl

Returns the largest integral value not greater than or equal to the given floating-point number. [Format] #include <math.h> double floor (double d); float floorf (float d); long double floorl (long double d); [Parameters] d Floating-point number for which largest integral value not greater than that number is to be computed [Return values] Largest integral value not greater than or equal to d [Remarks]

The floor function returns the largest integral value not greater than or equal to d, expressed as a double type value. Therefore, if d is negative, the value after rounding-up of the fractional part is returned.

fmod/fmodf/fmodl

```
Calculates the remainder of a division of two floating-point numbers.
[Format]
#include <math.h>
double fmod (double x, double y);
float fmodf (float x, float y);
long double fmodl (long double x, long double y);
[Parameters]
x Dividend
y Divisor
[Return values]
When y is 0.0: x
When y is not 0.0: Remainder of division of x by y
[Remarks]
In the fmod function, the relationship between parameters x and y and return value ret is as follows:
x = y * i + ret (where i is an integer)
The sign of return value ret is the same as the sign of x.
If the quotient of x/y cannot be represented, the value of the result is not guaranteed.
```

acosh/acoshf/acoshl

```
Calculates the hyperbolic arc cosine of a floating-point number.
[Format]
#include <math.h>
double acosh(double d);
float acoshf(float d);
long double acoshl(long double d);
[Parameters]
```



d Floating-point number for which hyperbolic arc cosine is to be computed [Return values]
Normal: Hyperbolic arc cosine of d
Abnormal: Domain error: Returns NaN.
Error conditions: A domain error occurs when d is smaller than 1.0.
[Remarks]
The acosh function returns the hyperbolic arc cosine in the range [0, +∞].

asinh/asinhf/asinhl

Calculates the hyperbolic arc sine of a floating-point number. [Format] #include <math.h> double asinh(double d); float asinhf(float d); long double asinhl(long double d); [Parameters] d Floating-point number for which hyperbolic arc sine is to be computed [Return values] Hyperbolic arc sine of d

atanh/atanhf/atanhl

Calculates the hyperbolic arc tangent of a floating-point number. [Format] #include <math.h> double atanh(double d); float atanhf(float d); long double atanhl(long double d); [Parameters] d Floating-point number for which hyperbolic arc tangent is to be computed [Return values] Normal: Hyperbolic arc tangent of d Abnormal: Domain error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL depending on the function. Range error: Returns not-a-number. [Remarks] A domain error occurs for a value of d not in the range [-1, +1]. A range error may occur for a value of d equal to -1 or 1.

exp2/exp2f/exp2l

Calculates the value of 2 raised to the power d. [Format] #include <math.h> double exp2(double d); float exp2f(float d); long double exp2l(long double d); [Parameters] d Floating-point number for which exponential function is to be computed [Return values] Normal: Exponential function value of 2 Abnormal: Range error: Returns 0, or returns +HUGE_VALF, or +HUGE_VALL depending on the function [Remarks] A range error occurs if the absolute value of d is too large.

expm1/expm1f/expm1l

Calculates the value of natural logarithm base e raised to the power d and subtracts 1 from the result. [Format]



#include <math.h>
double expm1(double d);
float expm1(float d);
long double expm1(long double d);
[Parameters]
d Power value to which natural logarithm base e is to be raised
[Return values]
Normal: Value obtained by subtracting 1 from natural logarithm base e raised to the power d
Abnormal: Range error: Returns -HUGE_VAL, -HUGE_VALF, or -HUGE_VALL depending on the function.
[Remarks]
expm1(d) provides more accurate calculation than exp(x) – 1 even when d is near to 0.

ilogb/ilogbf/ilogbl

Extracts the exponent of d. [Format] #include <math.h> long ilogb(double d); long ilogbf(float d); long ilogbl(long double d); [Parameters] d Value of which exponent is to be extracted [Return values] Normal:Exponential function value of d d is ∞: INT MAX d is not-a-number: FP ILOGBNAN d is 0: FP_ILOGBNAN Abnormal: d is 0 and a range error has occurred: FP_ILOGB0 [Remarks] A range error may occur if d is 0.

log1p/log1pf/log1pl

Calculates the natural logarithm (base e) of d + 1. [Format] #include <math.h> double log1p(double d); float log1pf(float d); long double log1pl(long double d); [Parameters] d Value for which the natural logarithm of this parameter + 1 is to be computed [Return values] Normal: Natural logarithm of d + 1 Abnormal: Domain error: Returns not-a-number. Range error: Returns –HUGE_VAL, –HUGE_VALF, or –HUGE_VALL depending on the function. [Remarks] A domain error occurs if d is smaller than -1. A range error occurs if d is -1. log1p(d) provides more accurate calculation than log(1+d) even when d is near to 0.

log2/log2f/log2l

Calculates the base-2 logarithm of d.
[Format]
#include <math.h>
double log2(double d);
float log2f(float d);
long double log21(long double d);
[Parameters]
d Value of which logarithm is to be calculated



[Return values] Normal: Base-2 logarithm of d Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if d is a negative value.

logb/logbf/logbl

Extracts the exponent of d in internal floating-point representation, as a floating-point value.

[Format]
#include <math.h>
double logb(double d);
float logbf(float d);
long double logbl(long double d);
[Parameters]
d Value of which exponent is to be extracted
[Return values]
Normal: Signed exponent of d
Abnormal: Range error: Returns -HUGE_VAL, -HUGE_VALF, or -HUGE_VALL depending on the function.
[Remarks]
A range error may occur if d is 0.
d is always assumed to be normalized.

scalbn/scalbnf/scalbnl/scalbln/scalblnf/scalblnl

Calculates a floating-point number multiplied by a power of radix, which is an integer. [Format] #include <math.h> double scalbn(double d, long e); float scalbnf(float d, long e); long double scalbnl(long double d, long e); double scalbln(double d, long e); float scalblnf(float d, long int e); long double scalblnl(long double d, long int e); [Parameters] d Value to be multiplied by FLT RADIX raised to the power e e Exponent used to compute a power of FLT RADIX [Return values] Normal: Value equal to d multiplied by FLT RADIX Abnormal: Range error: Returns -HUGE_VAL, -HUGE_VALF, or -HUGE_VALL depending on the function. [Remarks] A range error may occur if d is 0. FLT RADIX raised to the power e is not actually calculated.

cbrt/cbrtf/cbrtl

Calculates the cube root of a floating-point number. [Format] #include <math.h> double cbrt(double d); float cbrtf(float d); long double cbrtl(long double d); [Parameters] d Value for which a cube root is to be computed [Return values] Cube root of d

hypot/hypotf/hypotl



Calculates the square root of the sum of floating-point numbers raised to the power 2. [Format] #include <math.h> double hypot(double d, double e); float hypotf(float d, double e); long double hypotl(long double d, double e); [Parameters] d Values for which the square root of the sum of these values e raised to the power 2 is to be computed [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. [Remarks] A range error may occur if the result overflows.

erf/erff/erfl

Calculates the error function value of a floating-point number. [Format] #include <math.h> double erf(double d); float erff(float d); long double erfl(long double d); [Parameters] d Value for which the error function value is to be computed [Return values] Error function value of d

erfc/erfcf/erfcl

Calculates the complementary error function value of a floating-point number.

[Format]
#include <math.h>
double erfc(double d);
float erfcf(float d);
long double erfcl(long double d);
[Parameters]
d Value for which the complementary error function value is to be computed
[Return values]
Complementary error function value of d
[Remarks]
A range error occurs if the absolute value of d is too large.

lgamma/lgammaf/lgammal

Calculates the logarithm of the gamma function of a floating-point number. [Format] #include <math.h> double lgamma(double d); float lgammaf(float d); long double lgammal(long double d); [Parameters] d Value for which the logarithm of the gamma function is to be computed [Return values] Normal: Logarithm of gamma function of d Abnormal: Domain error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL with the mathematically correct sign. Range error: Returns +HUGE_VAL, +HUGE_VALF, or +HUGE_VALL. [Remarks] A range error is set if the absolute value of d is too large or small. A domain error occurs if d is a negative integer or 0 and the calculation result is not representable.



tgamma/tgammaf/tgammal

Calculates the gamma function of a floating-point number.

[Format]
#include <math.h>
double tgamma(double d);
float tgamma(float d);
long double tgammal(long double d);
[Parameters]
d Value for which the gamma function value is to be computed
[Return values]
Normal: Gamma function value of d
Abnormal: Domain error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL with the same sign as that of d.
Range error: Returns 0, or returns +HUGE_VAL, +HUGE_VALF, or +HUGE_VALL with the mathematically correct sign
depending on the function.
[Remarks]
A range error is set if the absolute value of d is too large or small.

A domain error occurs if d is a negative integer or 0 and the calculation result is not representable.

nearbyint/nearbyintf/nearbyintl

Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.

[Format]
#include <math.h>
double nearbyint(double d);
float nearbyint(float d);
long double nearbyintl(long double d);
[Parameters]
d Value to be rounded to an integer in the floating-point format
[Return values]
d rounded to an integer in the floating-point format
[Remarks]
The nearbyint function group does not generate "inexact" floating-point exceptions.

rint/rintf/rintl

Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.

```
[Format]
#include <math.h>
double rint(double d);
float rintf(float d);
long double rintl(long double d);
[Parameters]
d Value to be rounded to an integer in the floating-point format
[Return values]
d rounded to an integer in the floating-point format
[Remarks]
The rint function group differs from the pearbyint function group
```

The rint function group differs from the nearbyint function group only in that the ring function group may generate "inexact" floating-point exceptions.

lrint/lrintf/lrintl/llrint/llrintf/llrintl

Rounds a floating-point number to the nearest integer according to the current rounding direction.

```
[Format]
#include <math.h>
long int lrint(double d);
```



long int lrintf(float d); long int lrintl(long double d); long long int llrint(double d); long long int llrintf(float d); long long int llrintl(long double d); [Parameters] d Value to be rounded to an integer [Return values] Normal: d rounded to an integer Abnormal: Range error: Returns an undetermined value. [Remarks] A range error may occur if the absolute value of d is too large. The return value is unspecified when the rounded value is not in the range of the return value type.

round/roundf/roundl/lroundf/lroundf/lroundl/llroundf/llroundl

Rounds a floating-point number to the nearest integer.

```
[Format]
```

```
#include <math.h>
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);
[Parameters]
d Value to be rounded to an integer
[Return values]
Normal: d rounded to an integer
Abnormal: Range error: Returns an undetermined value.
[Remarks]
```

A range error may occur if the absolute value of d is too large.

When d is at the midpoint between two integers, the Iround 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.

trunk/truncf/truncl

Rounds a floating-point number to the nearest integer in the floating-point representation.

```
[Format]
#include <math.h>
double trunc(double d);
float truncf(float d);
long double truncl(long double d);
[Parameters]
d Value to be rounded to an integer in the floating-point representation
[Return values]
d truncated to an integer in the floating-point format
[Remarks]
The trunc function group rounds d so that the absolute value after rounding is not greater than the absolute value of d.
```

remainder/remainderf/remainderl

Calculates the remainder of a division of two floating-point numbers.
[Format]
#include <math.h>
double remainder(double d1, double d2);



float remainderf(float d1, float d2); long double remainderl(long double d1, long double d2); [Parameters] d1, d2 Values for which remainder of a division is to be computed (d1 / d2) [Return values] Remainder of division of d1 by d2 [Remarks] The remainder calculation by the remainder function group conforms to the IEEE 60559 standard.

remquo/remquof/remquol

Calculates the remainder of a division of two floating-point numbers. [Format] #include <math.h> double remquo(double d1, double d2, long *q); float remquof(float d1, float d2, long *q); long double remquol(long double d1, long double d2, long *q); [Parameters] d1, d2 Values for which remainder of a division is to be computed (d1 / d2) q Value pointing to the location to store the quotient obtained by remainder calculation [Return values] Remainder of division of d1 by d2 [Remarks] The value stored in the location indicated by g has the same sign as the result of x/y and the

The value stored in the location indicated by q has the same sign as the result of x/y and the integral quotient of modulo-2n x/y (n is an implementation-defined integer equal to or greater than 3).

copysign/copysignf/copysignl

Generates a value consisting of the absolute value of d1 and the sign of d2.
[Format]
#include <math.h>
double copysign(double d1, double d2);
float copysignf(float d1, float d2);
long double copysignl(long double d1, long double d2);
[Parameters]
d1 Value of which absolute value is to be used in the generated value
d2 Value of which sign is to be used in the generated value

[Return values] Normal: Value consisting of absolute value of d1 and sign of d2 Abnormal: Range error: Returns an undetermined value. [Remarks] When d1 is a not-a-number, the copysign function group generates a not-a-number with the sign bit of d2.

nan/nanf/nanl

Returns not-a-number. [Format] #include <math.h> double nan(const char *c); float nanf(const char *c); long double nanl(const char *c); [Parameters] c Pointer to a string [Return values] qNaN with the contents of the location indicated by c or 0 (when qNaN is not supported) [Remarks] The nan("c string") call is equivalent to strtod("NAN(c string)", (char**) NULL). The nanf and nanl calls are equivalent to

the corresponding strtof and strtold calls, respectively.



nextafter/nextafterf/nextafterl

Calculates the next floating-point representation following d1 in the direction to d2 on the real axis. [Format] #include <math.h> double nextafter(double d1, double d2); float nextafterf(float d1, float d2); long double nextafterl(long double d1, long double d2); [Parameters] d1 Floating-point value on the real axis d2 Value indicating the direction viewed from d1, in which a representable floating-point value is to be found [Return values] Normal: Representable floating-point value Abnormal: Range error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL with the mathematically correct sign depending on the function.

[Remarks]

A range error may occur if d1 is the maximum finite value that can be represented in its type and the return value is an infinity or cannot be represented in its type.

The nextafter function group returns d2 when d1 is equal to d2.

nexttoward/nexttowardf/nexttowardl

Calculates the next floating-point representation following d1 in the direction to d2 on the real axis.

[Format]

#include <math.h>

double nexttoward(double d1, long double d2);

float nexttowardf(float d1, long double d2); long double nexttowardl(long double d1, long double d2);

[Parameters]

d1 Floating-point value on the real axis

d2 Value indicating the direction viewed from d1, in which a representable floating-point value is to be found [Return values]

Normal: Representable floating-point value

Abnormal: Range error: Returns HUGE_VAL, HUGE_VALF, or HUGE_VALL with the mathematically correct sign depending on the function

[Remarks]

A range error may occur if d1 is the maximum finite value that can be represented in its value and the return value is an infinity or cannot be represented in its type.

The nexttoward function group is equivalent to the nextafter function group except that d2 is of type long double and returns d2 after conversion depending of the function when d1 is equal to d2.

fdim/fdimf/fdiml

Calculates the positive difference between two arguments. [Format] #include <math.h> double fdim(double d1, double d2); float fdimf(float d1, float d2); long double fdiml(long double d1, long double d2); [Parameters] d1, d2 Values of which difference is to be computed (|d1 - d2|) [Return values] Normal: Positive difference between two arguments Abnormal: Range error: HUGE_VAL, HUGE_VALF, or HUGE_VALL [Remarks] A range error may occur if the return value overflows.

fmax/fmaxf/fmaxl



```
Obtains the greater of two arguments.

[Format]

#include <math.h>

double fmax(double d1, double d2);

float fmaxf(float d1, float d2);

long double fmaxl(long double d1, long double d2);

[Parameters]

d1, d2 Values to be compared

[Return values]

Greater of two arguments

[Remarks]

The fmax function group recognizes a not-a-number as a lack of data
```

The fmax function group recognizes a not-a-number as a lack of data. When one argument is a not-a-number and the other is a numeric value, the function returns the numeric value.

fmin/fminf/fminl

Obtains the smaller of two arguments.

```
[Format]
#include <math.h>
double fmin(double d1, double d2);
float fminf(float d1, float d2);
long double fminl(long double d1, long double d2);
[Parameters]
d1, d2 Values to be compared
[Return values]
Smaller of two arguments
[Remarks]
The fmin function group recognizes a not-a-number as a lack of data. V
```

The fmin function group recognizes a not-a-number as a lack of data. When one argument is a not-a-number and the other is a numeric value, the function returns the numeric value.

fma/fmaf/fmal

```
Calculates (d1 * d2) + d3 as a single ternary operation.
[Format]
#include <math.h>
double fma(double d1, double d2, double d3);
float fmaf(float d1, float d2, float d3);
long double fmal(long double d1, long double d2, long double d3);
[Return values]
Result of (d1 * d2) + d3 calculated as ternary operation
[Parameters]
d1, d2, d3 Floating-point values
[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.



6.4.8 <mathf.h>

Performs various mathematical operations.

<mathf.h> declares mathematical functions and defines macros in single-precision format. The mathematical functions and macros used here do not follow the ANSI specifications. Each function receives **float**-type arguments and returns a **float**-type value.

The following constants (macros) are all implementation-defined.

Туре	Definition Name	Description
Constant (macro)	EDOM	Indicates the value to be set in errno if the value of a parameter input to a func- tion 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 repre- sented 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 over- flows.
Function	acosf	Calculates the arc cosine of a floating-point number.
	asinf	Calculates the arc sine of a floating-point number.
	atanf	Calculates the arc tangent of a floating-point number.
	atan2f	Calculates the arc tangent of the result of a division of two floating-point numbers.
	cosf	Calculates the cosine of a floating-point radian value.
	sinf	Calculates the sine of a floating-point radian value.
	tanf	Calculates the tangent of a floating-point radian value.
	coshf	Calculates the hyperbolic cosine of a floating-point number.
	sinhf	Calculates the hyperbolic sine of a floating-point number.
	tanhf	Calculates the hyperbolic tangent of a floating-point number.
	expf	Calculates the exponential function of a floating-point number.
	frexpf	Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.
	ldexpf	Multiplies a floating-point number by a power of 2.
	logf	Calculates the natural logarithm of a floating-point number.
	log10f	Calculates the base-ten logarithm of a floating-point number.
	modff	Breaks a floating-point number into integral and fractional parts.
	powf	Calculates a power of a floating-point number.
	sqrtf	Calculates the positive square root of a floating-point number.
	ceilf	Calculates the smallest integral value not less than or equal to the given float- ing-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 float- ing-point number.
	fmodf	Calculates the remainder of a division of two floating-point numbers.

Operation in the event of an error is described below.

(1) Domain error



A domain error occurs if the value of a parameter input to a function is outside the domain over which the mathematical function is defined. In this case, the value of **EDOM** is set in **errno**. The function return value in implementation-defined.

(2) Range error

A range error occurs if the result of a function cannot be represented as a **float** type value. In this case, the value of **ERANGE** is set in **errno**. If the result overflows, the function returns the value of **HUGE_VALF**, with the same sign as the correct value of the function. If the result underflows, 0 is returned as the return value.

Notes 1. If there is a possibility of a domain error resulting from a **<mathf.h>** function call, it is dangerous to use the resultant value directly. The value of **errno** should always be checked before using the result in such cases.

[Format]

```
:

1 x=asinf(a);

2 if (errno==EDOM)

3 printf ("error\n");

4 else

5 printf ("result is : %f\n",x);

.

.
```

In line 1, the arc sine value is computed using the **asinf** function. If the value of argument **a** is outside the **asinf** function domain [-1.0, 1.0], the **EDOM** value is set in **errno**. Line 2 determines whether a domain error has occurred. If a domain error has occurred, error is output in line 3. If there is no domain error, the arc sine value is output in line 5.

Notes 2. Whether or not a range error occurs depends on the internal representation format of floating-point types determined by the compiler. For example, if an internal representation format that allows an infinity to be represented as a value is used, **<mathf.h>** library functions can be implemented without causing range errors.

Implementation-Defined Specifications

Item	Compiler Specifications
Value returned by a mathematical function if an input argument is out of the range	A not-a-number is returned. For details on the format of not-a-numbers, refer to section 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.

acosf

Calculates the arc cosine of a floating-point number. [Format] #include <mathf.h> float acosf (float f); [Parameters] f Floating-point number for which arc cosine is to be computed [Return values] Normal: Arc cosine of f Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs for a value of f not in the range [-1.0, +1.0] The acosf function returns the arc cosine in the range [0, π] by the radian.



asinf

Calculates the arc sine of a floating-point number. [Format] #include <mathf.h> float asinf (float f); [Parameters] f Floating-point number for which arc sine is to be computed [Return values] Normal: Arc sine of f Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs for a value of f not in the range [-1.0, +1.0]. The asinf function returns the arc sine in the range [-π/2, +π/2] by the radian.

atanf

Calculates the arc tangent of a floating-point number. [Format] #include <mathf.h> float atanf (float f); [Parameters] f Floating-point number for which arc tangent is to be computed [Return values] Arc tangent of f [Remarks] The atanf function returns the arc tangent in the range $(-\pi/2, +\pi/2)$ by the radian.

atan2f

Calculates the arc tangent of the division of two floating-point numbers. [Format] #include <mathf.h> float atan2f (float y, float x); [Parameters] x Divisor y Dividend [Return values] Normal: Arc tangent value when y is divided by x Abnormal: Domain error: Returns not-a-number. Error conditions: A domain error occurs if the values of both x and y are 0.0. [Remarks] A domain error occurs if the values of both x and y are 0.0.

The atan2f function returns the arc tangent in the range $(-\pi, +\pi)$ by the radian. The meaning of the atan2f function is illustrated in figure 6.2. As shown in the figure, the result of the atan2f function is the angle between the X-axis and a straight line passing through the origin and point (x, y).

If y = 0.0 and x is negative, the result is π . If x = 0.0, the result is $\pm \pi/2$, depending on whether y is positive or negative.

Figure 6.2 atan2f 関数の意味





cosf

Calculates the cosine of a floating-point radian value. [Format] #include <mathf.h> float cosf (float f); [Parameters] f Radian value for which cosine is to be computed [Return values] Cosine of f

sinf

Calculates the sine of a floating-point radian value.
[Format]
#include <mathf.h>
float sinf (float f);
[Parameters]
f Radian value for which sine is to be computed
[Return values]
Sine of f

tanf

Calculates the tangent of a floating-point radian value. [Format] #include <mathf.h> float tanf (float f); [Parameters] f Radian value for which tangent is to be computed [Return values] Tangent of f

coshf

Calculates the hyperbolic cosine of a floating-point number. [Format] #include <mathf.h> float coshf (float f); [Parameters] f Floating-point number for which hyperbolic cosine is to be computed [Return values] Hyperbolic cosine of f

sinhf

Calculates the hyperbolic sine of a floating-point number. [Format] #include <mathf.h> float sinhf (float f); [Parameters] f Floating-point number for which hyperbolic sine is to be computed [Return values] Hyperbolic sine of f



tanhf

Calculates the hyperbolic tangent of a floating-point number. [Format] #include <mathf.h> float tanhf (float f); [Parameters] f Floating-point number for which hyperbolic tangent is to be computed [Return values] Hyperbolic tangent of f

expf

Calculates the exponential function of a floating-point number. [Format] #include <mathf.h> float expf (float f); [Parameters] f Floating-point number for which exponential function is to be computed [Return values] Exponential function value of f

frexpf

Breaks a floating-point number into a [0.5, 1.0) value and a power of 2. [Format] #include <mathf.h> float frexpf (float value, float long *exp); [Parameters] value Floating-point number to be broken into a [0.5, 1.0) value and a power of 2 exp Pointer to storage area that holds power-of-2 value [Return values] If value is 0.0: 0.0 If value is not 0.0: Value of ret defined by ret * 2^{value pointed to by exp} = value [Remarks] The frexpf function breaks value into a [0.5, 1.0) value and a power of 2. It stores the resultant power-of-2 value in the area pointed to by exp. The frexpf function returns the return value ret in the range [0.5, 1.0) or as 0.0. If value is 0.0, the contents of the int storage area pointed to by exp and the value of ret are both 0.0.

ldexpf

Multiplies a floating-point number by a power of 2. [Format] #include <mathf.h> float ldexpf (float e, long f); [Parameters] e Floating-point number to be multiplied by a power of 2 f Power-of-2 value [Return values] Result of e * 2^f operation

logf

Calculates the natural logarithm of a floating-point number.
[Format]
#include <mathf.h>
float logf (float f);
[Parameters]



f Floating-point number for which natural logarithm is to be computed [Return values] Normal: Natural logarithm of f Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if f is negative. A range error occurs if f is 0.0.

log10f

Calculates the base-ten logarithm of a floating-point number. [Format] #include <mathf.h> float log10f (float f); [Parameters] f Floating-point number for which base-ten logarithm is to be computed [Return values] Normal: Base-ten logarithm of f Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if f is negative. A range error occurs if f is 0.0.

modff

Breaks a floating-point number into integral and fractional parts. [Format] #include <mathf.h> float modff (float a, float *b); [Parameters] a Floating-point number to be broken into integral and fractional parts b Pointer indicating storage area that stores integral part [Return values] Fractional part of **a**

powf

Calculates a power of a floating-point number. [Format] #include <mathf.h> float powf (float x, float y); [Parameters] x Value to be raised to a power y Power value [Return values] Normal: Value of x raised to the power y Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if x is 0.0 and y is 0.0 or less, or if x is negative and y is not an integer.

sqrtf

Calculates the positive square root of a floating-point number.
[Format]
#include <mathf.h>
float sqrtf (float f);
[Parameters]
f Floating-point number for which positive square root is to be computed
[Return values]



Normal: Positive square root of f Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if f is negative.

ceilf

Returns the smallest integral value not less than or equal to the given floating-point number. [Format] #include <mathf.h> float ceilf (float f); [Parameters] f Floating-point number for which smallest integral value not less than that number is to be computed [Return values] Smallest integral value not less than or equal to f [Remarks] The ceilf function returns the smallest integral value not less than or equal to f, expressed as a float type value. Therefore, if f is negative, the value after truncation of the fractional part is returned.

fabsf

Calculates the absolute value of a floating-point number. [Format] #include <mathf.h> float fabsf (float f); [Parameters] f Floating-point number for which absolute value is to be computed [Return values] Absolute value of f

floorf

Returns the largest integral value not greater than or equal to the given floating-point number. [Format] #include <mathf.h> float floorf (float f); [Parameters] f Floating-point number for which largest integral value not greater than that number is to be computed [Return values] Largest integral value not greater than or equal to f [Remarks] The floorf function returns the largest integral value not greater than or equal to f, expressed as a float type value. Therefore, if f is negative, the value after rounding-up of the fractional part is returned.

fmodf

Calculates the remainder of a division of two floating-point numbers. [Format] #include <mathf.h> float fmodf (float x, float y); [Parameters] x Dividend y Divisor [Return values] When y is 0.0: x When y is not 0.0: Remainder of division of x by y [Remarks] In the fmodf function, the relationship between parameters x and y and return value ret is as follows: x = y * i + ret (where i is an integer)

RENESAS

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.

6.4.9 <setjmp.h>

Supports transfer of control between functions. The following macros are implementation-defined.

Туре	Definition Name	Description
Type (macro)	jmp_buf	Indicates the type name corresponding to a storage area for storing information that enables transfer of control between functions.
Function	setjmp	Saves the execution environment defined by jmp_buf of the currently executing function in the specified storage area.
	longjmp	Restores the function execution environment saved by the setjmp function, and transfers control to the program location at which the setjmp function was called.

The **setjmp** function saves the execution environment of the current function. The location in the program that called the **setjmp** function can subsequently be returned to by calling the **longjmp** function.

An example of how transfer of control between functions is supported using the **setjmp** and **longjmp** functions is shown below.

```
[Format]
```

```
1
     #include <stdio.h>
     #include <setjmp.h>
2
3
     jmp_buf env;
4
     void sub( );
5
     void main( )
6
     {
7
8
              if (setjmp(env)!=0){
9
                   printf("return from longjmp\n");
10
                   exit(0);
               }
11
12
               sub( );
13
     }
14
15
      void sub( )
16
     {
17
             printf("subroutine is running \n");
18
             longjmp(env, 1);
19
     }
```

Explanation:

The **setjmp** function is called in line 8. At this time, the environment in which the **setjmp** function was called is saved in **jmp_buf** type variable **env**. The return value in this case is 0, and therefore function **sub** is called next.

The environment saved in variable **env** is restored by the **longjmp** function called within function **sub**. As a result, the program behaves just as if a return had been made from the **setjmp** function in line 8. However, the return value at this time is 1 specified by the second argument of the **longjmp** function. As a result, execution proceeds to line 9.

setjmp

Saves the execution environment of the currently executing function in the specified storage area. [Format] #include <setjmp.h> long setjmp (jmp_buf env); [Parameters] env Pointer to storage area in which execution environment is to be saved [Return values] When setjmp function is called: 0



On return from **longjmp** function: Nonzero

[Remarks]

The execution environment saved by the **setjmp** function is used by the **longjmp** function. The return value is 0 when the function is called as the **setjmp** function, but the return value on return from the **longjmp** function is the value of the second parameter specified by the **longjmp** function.

If the **setjmp** function is called from a complex expression, part of the current execution environment, such as the intermediate result of expression evaluation, may be lost. The **setjmp** function should only be used in the form of a comparison between the result of the **setjmp** function and a constant expression, and should not be called within a complex expression.

Do not call the **setjmp** function indirectly using a pointer.

longjmp

Restores the function execution environment saved by the setimp function, and transfers control to the program location at which the setimp function was called.

[Format]

#include <setjmp.h>
void longjmp (jmp_buf env, long ret);
[Parameters]
env Pointer to storage area in which execution environment was saved
ret Return code to setjmp function

[Remarks]

From the storage area specified by the first parameter env, the longjmp function restores the function execution environment saved by the most recent invocation of the setjmp function in the same program, and transfers control to the program location at which that setjmp function was called. The value of the second parameter ret of the longjmp function is returned as the setjmp function return value. However, if ret is 0, the value 1 is returned to the setjmp function as a return value.

If the setjmp function has not been called, or if the function that called the setjmp function has already executed a return statement, the operation of the longjmp function is not guaranteed.

6.4.10 <stdarg.h>

Enables referencing of variable arguments for functions with such arguments. The following macros are implementation-defined.

Туре	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 referenc- ing.
	va_arg	Enables referencing of the argument following the argument currently being referenced for a function with variable arguments.
	va_end	Terminates referencing of the arguments of a function with variable arguments.
	va_copy <-lang=c99>	Copies variable arguments.

An example of a program using the macros defined by this standard include file is shown below.

```
[Format]
     #include <stdio.h>
1
2
     #include <stdarg.h>
3
 4
     extern void prlist(int count, ...);
5
6
     void main( )
7
    {
8
        prlist(1, 1);
        prlist(3, 4, 5, 6);
9
        prlist(5, 1, 2, 3, 4, 5);
10
11
     }
```

12



```
13
     void prlist(int count, ...)
14
    {
15
        va_list ap;
        int i;
16
17
18
        va_start(ap, count);
19
        for(i=0; i<count; i++)</pre>
20
             printf("%d", va_arg(ap, int));
21
        putchar('\n');
22
        va_end(ap);
   }
23
```

Explanation:

This example implements function **prlist**, in which the number of data items to be output is specified in the first argument and that number of subsequent arguments are output.

In line 18, the variable argument reference is initialized by va_start. Each time an argument is output, the next argument is referenced by the va_arg macro (line 20). In the va_arg macro, the type name of the argument (in this case, int type) is specified in the second argument.

When argument referencing ends, the va_end macro is called (line 22).

va	start

Executes initialization processing for referencing variable arguments.

[Format]

#include <stdarg.h>

void va_start (va_list ap, parmN)

[Parameters]

ap Variable for accessing variable arguments

parmN Identifier of rightmost argument

[Remarks]

The va_start macro initializes ap for subsequent use by the va_arg and va_end macros.

The argument parmN is the identifier of the rightmost argument in the argument list in the external function definition (the one just before the , ...).

To reference variable unnamed arguments, the va_start macro call must be executed first of all.

va	arq
	_

Allows a reference to the argument following the argument currently being referred to in the function with variable arguments.

[Format]

```
#include <stdarg.h>
type va_arg (va_list ap, type);
[Parameters]
ap Variable for accessing variable arguments
type Type of arguments to be accessed
[Return values]
Argument value
[Remarks]
```

Specify a variable of the va_list type initialized by the va_start macro as the first argument. The value of ap is updated each time va_arg is used, and, as a result, a sequence of variable arguments is returned by sequential calls of this macro. Specify the type to refer to as the second argument type.

The ap argument must be the same as the ap initialized by va_start.

It will not be possible to refer to arguments correctly if argument type is set to a type of which size is changed by type conversion when it is used as a function argument, i.e., if char type, unsigned char type, short type, unsigned short type, or float type is specified as type. If such a type is specified, correct operation is not guaranteed.

va_end

Terminates referencing of the arguments of a function with variable arguments. [Format]



```
#include <stdarg.h>
void va_end (va_list ap);
[Parameters]
ap Variable for referencing variable arguments
[Remarks]
```

The **ap** argument must be the same as the **ap** initialized by **va_start**. If the **va_end** macro is not called before the return from a function, the operation of that function is not guaranteed.

va_copy

Makes a copy of the argument currently being referenced for a function with variable arguments.

```
[Format]
#include <stdarg.h>
void va_copy (va_list dest, va_list src);
[Parameters]
dest Copy of variable for referencing variable arguments
src Variable for referencing variable arguments
[Remarks]
```

A copy is made of the second argument src which is one of the variable arguments that have been initialized by the va_start macro and used by the va_arg macro, and the copy is saved in the first argument dest.

The src argument must be the same as the src initialized by va_start.

The dest argument can be used as an argument that indicates the variable arguments in the subsequent va_arg macros.

6.4.11 <stdio.h>

Performs processing relating to input/output of stream input/output file. The following constants (macros) are all implementation-defined.


Туре	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.		
	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.		
Function	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 for- mat.		



Туре	Definition Name Description			
	snprintf <-lang=c99>	Converts data according to a format and writes it to the specified array.		
	vsnprintf <-lang=c99>	Equivalent to snprintf with the variable argument list replaced by va_list .		
	vfscanf <-lang=c99>	Equivalent to fscanf with the variable argument list replaced by va_list .		
	vscanf <-lang=c99>	Equivalent to scanf with the variable argument list replaced by va_list .		
	vsscanf <-lang=c99>	Equivalent to sscanf with the variable argument list replaced by va_list .		
	fscanf	Inputs data from a stream input/output file and converts it according to a for- mat.		
	scanf	Inputs data from the standard input file (stdin) and converts it according to a format.		
	vsprintf	Outputs a variable parameter list to the specified area according to a format.		
	fgetc	Inputs one character from a stream input/output file.		
	fgets	Inputs a string from a stream input/output file.		
	fputc	Outputs one character to a stream input/output file.		
	fputs	Outputs a string to a stream input/output file.		
	getc	(macro) Inputs one character from a stream input/output file.		
	getchar	(macro) Inputs one character from the standard input file.		
	gets	Inputs a string from the standard input file.		
	putc	(macro) Outputs one character to a stream input/output file.		
	putchar	(macro) Outputs one character to the standard output file.		
	puts	Outputs a string to the standard output file.		
	ungetc	Returns one character to a stream input/output file.		
	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 begin- ning 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).		
Туре	fpos_t	Indicates a type that can specify any position in a file.		
Constant	FOPEN_MAX	Indicates the maximum number of files that can be opened simultaneously.		
(macro)	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	Hexadecimal representation.
The meaning of conversion specifier '-' in the fscanf function	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 6.7 shows the format when displaying the floating-point infinity and not-a-number in **printf** and **fprintf** functions.

Table 6.7	Display Format of Infinity and Not-a-Number
-----------	---

Value	Display Format
Positive infinity	+++++
Negative infinity	
Not-a-number	****

An example of a program that performs a series of input/output processing operations for a stream input/output file is shown in the following.

```
[Format]
1
    #include <stdio.h>
2
3
  void main( )
4
  {
5
       int c;
6
       FILE *ifp, *ofp;
7
8
       if ((ifp=fopen("INPUT.DAT","r"))==NULL){
9
           fprintf(stderr,"cannot open input file\n");
           exit(1);
10
```



```
11
        }
        if ((ofp=fopen("OUTPUT.DAT","w"))==NULL){
12
            fprintf(stderr,"cannot open output file\n");
13
14
            exit(1);
15
        }
16
        while ((c=getc(ifp))!=EOF)
17
           putc(c, ofp);
18
        fclose(ifp);
19
        fclose(ofp);
20
   }
```

Explanation:

This program copies the contents of file **INPUT.DAT** to file **OUTPUT.DAT**.

Input file **INPUT.DAT** is opened by the **fopen** function in line 8, and output file **OUTPUT.DAT** is opened by the **fopen** function in line 12. If opening fails, **NULL** is returned as the return value of the **fopen** function, an error message is output, and the program is terminated.

If the **fopen** function ends normally, the pointer to the data (**FILE** type) that stores information on the opened files is returned; these are set in variables **ifp** and **ofp**.

After successful opening, input/output is performed using these **FILE** type data. When file processing ends, the files are closed with the **fclose** function.

fclose

Closes a stream input/output file.

[Format]
#include <stdio.h>
long fclose (FILE *fp);
[Parameters]
fp File pointer
[Return values]
Normal: 0
Abnormal: Nonzero
[Remarks]
The fclose function closes the stream input/output file indicated by file pointer fp.
If the output file of the stream input/output file is open and data that is not output remains in the buffer, that data is output

to the file before it is closed.

If the input/output buffer was automatically allocated by the system, it is released.

fflush

Outputs the stream input/output file buffer contents to the file.

[Format]
#include <stdio.h>
long fflush (FILE *fp);
[Parameters]
fp File pointer
[Return values]
Normal: 0
Abnormal: Nonzero
[Remarks]

When the output file of the stream input/output file is open, the fflush function outputs the contents of the buffer that is not output for the stream input/output file specified by file pointer fp to the file. When the input file is open, the ungetc function specification is invalidated.

fopen

Opens a stream input/output file under the specified file name.
[Format]
#include <stdio.h>
FILE *fopen (const char *fname, const char *mode);
[Parameters]

RENESAS

fname Pointer to string indicating file name

mode Pointer to string indicating file access mode

[Return values]

Normal: File pointer indicating file information on opened file

Abnormal: NULL

[Remarks]

The fopen function opens the stream input/output file whose file name is the string pointed to by fname. If a file that does not exist is opened in write mode or append mode, a new file is created wherever possible. When an existing file is opened in write mode, writing processing is performed from the beginning of the file, and previously written file contents are erased.

When a file is opened in append mode, write processing is performed from the end-of-file position. When a file is opened in update mode, both input and output processing can be performed on the file. However, input cannot directly follow output without intervening execution of the fflush, fseek, or rewind function. Similarly, output cannot directly follow input without intervening execution of the fflush, fseek, or rewind function.

A string indicating the opening method may be added after the string indicating the file access mode.

freopen

Closes a currently open stream input/output file and reopens a new file under the specified file name. [Format]

#include <stdio.h>
#include <stdio.h>
FILE *freopen (const char *fname, const char *mode, FILE *fp);
[Parameters]
fname Pointer to string indicating new file name
mode Pointer to string indicating file access mode
fp File pointer to currently open stream input/output file
[Return values]
Normal: fp
Abnormal: NULL
[Remarks]

The freopen function first closes the stream input/output file indicated by file pointer fp (the following processing is carried out even if this close processing is unsuccessful). Next, the freopen function opens the file indicated by file name fname for stream input/output, reusing the FILE structure pointed to by fp.

The freopen function is useful when there is a limit on the number of files being opened at one time.

The freopen function normally returns the same value as fp, but returns NULL when an error occurs.

```
setbuf
```

Defines and sets a stream input/output buffer area by the user program.
[Format]
#include <stdio.h>
void setbuf (FILE *fp, char buf[BUFSIZ]);
[Parameters]
fp File pointer
buf Pointer to buffer area
[Remarks]

The setbuf function defines the storage area pointed to by buf so that it can be used as an input/output buffer area for the stream input/output file indicated by file pointer fp. As a result, input/output processing is performed using a buffer area of size BUFSIZ.

setvbuf

Defines and sets a stream input/output buffer area by the user program.
[Format]
#include <stdio.h>
long setvbuf (FILE *fp, char *buf, long type, size_t size);
[Parameters]
fp File pointer
buf Pointer to buffer area
type Buffer management method



size Size of buffer area [Return values] Normal: 0 Abnormal: Nonzero [Remarks]

The setvbuf function defines the storage area pointed to by buf so that it can be used as an input/output buffer area for the stream input/output file indicated by file pointer fp.

There are three ways of using this buffer area, as follows:

(a) When _IOFBF is specified as type

Input/output is fully buffered.

(b) When _IOLBF is specified as type

Input/output is line buffered; that is, input/output data is fetched from the buffer area when a new-line character is written, when the buffer area is full, or when input is requested.

(c) When _IONBF is specified as type

Input/output is unbuffered.

The setvbuf function usually returns 0. However, when an illegal value is specified for type or size, or when the request on how to use the buffer could not be accepted, a value other than 0 is returned.

The buffer area must not be released before the open stream input/output file is closed. In addition, the setvbuf function must be used between opening of the stream input/output file and execution of input/output processing.

fprintf

Outputs data to a stream input/output file according to the format.

[Format]
#include <stdio.h>
long fprintf (FILE *fp, const char *control[, arg]...);
[Parameters]
fp File pointer
control Pointer to string indicating format
arg,... List of data to be output according to format
[Return values]
Normal: Number of characters converted and output
Abnormal: Negative value
[Remarks]
The fprintf function converts and edits parameter arg according to the string that represents the format pointed to by
control, and outputs the result to the stream input/output file indicated by file pointer fp.

The fprintf function returns the number of characters converted and output when the function is terminated successfully, or a negative value if an error occurs.

The format specifications are shown below.

Overview of Formats

The string that represents the format is made up of two kinds of string.

- Ordinary characters

A character other than a conversion specification shown below is output unchanged.

- Conversion specifications

A conversion specification is a string beginning with % that specifies the conversion method for the following parameter. The conversion specifications format conforms to the following rules:

 $\label{eq:Flag...} & \left\{ \begin{bmatrix} \star \\ - \end{bmatrix} \\ [Field width] \end{array} \right\} \left(\star \left\{ \begin{bmatrix} \star \\ - \end{bmatrix} \\ [Precision] \end{array} \right\} \right) [Parameter size specification] Conversion specifier$

When there is no parameter to be actually output according to this conversion specification, the behavior is not guaranteed. In addition, when the number of parameters to be actually output is greater than the conversion specification, the excess parameters are ignored.

Description of Conversion Specifications

(a) Flags

Flags specify modifications to the data to be output, such as addition of a sign. The types of flag that can be specified and their meanings are shown in table 6.8.



Table 6.8	Flag Types and Their Meanings	
-----------	-------------------------------	--

Туре	Meaning
-	If the number of converted data characters is less than the field width, the data will be output left-justi- fied 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 6.10. 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 6.10.

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 $\boldsymbol{d},\,\boldsymbol{i},\,\boldsymbol{o},\,\boldsymbol{u},\,\boldsymbol{x},$ and \boldsymbol{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 6.10), the size (short type, long type, long long type, or long double type) of the data to be converted is specified. In other conversions, this specification is ignored. Table 6.9 shows the types of size specification and their meanings.

Table 6.9Parameter Size Specification Types and Meanings

Туре	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.
I	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.

II	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 6.10 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.

Conver- sion Speci- fier	Conversion Type	Conversion Method	Data Type Subject to Conver- sion	Notes on Precision
d	d conver- sion	int type data is converted to a signed decimal string. d conversion and i	int type	The precision specification indi- cates the minimum number of
i	i conversion	tion.	int type	converted data characters is less
0	o conver- sion	int type data is converted to an unsigned octal string.	int type	the string is prefixed with zeros. If the precision is omitted, 1 is
u	u conver- sion	int type data is converted to an unsigned decimal string.	int type	assumed. If conversion and output of data with a value of 0 is attempted with 0 specified as the
x	x conver- sion	int type data is converted to unsigned hexadecimal. a, b, c, d, e, and f are used as hexadecimal char- acters.	int type	precision, nothing will be output.
x	X conver- sion	int type data is converted to unsigned hexadecimal. A, B, C, D, E, and F are used as hexadecimal char- acters.	int type	
f	f conversion	double type data is converted to a decimal string with the format [–] ddd.ddd.	double type	The precision specification indi- cates 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 preci- sion is omitted, 6 is assumed. When 0 is specified as the preci- sion, the decimal point and subse- quent characters are not output. The output data is rounded.

Table 6.10 Conversion Specifiers and Conversion Methods



Conver- sion Speci- fier	Conversion Type	Conversion Method	Data Type Subject to Conver- sion	Notes on Precision
e	e conver- sion	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 indi- cates the number of digits after the decimal point. The format is such that one digit is output before the
E	E conver- sion	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	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 preci- sion, characters after the decimal point are not output. The output data is rounded.
g	g conver- sion (or G	Whether f conversion format output or e conversion (or E conversion) for-	double type	The precision specification indi- cates the maximum number of sig-
G	- conversion)	mat output is performed is deter- mined 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 preci- sion that indicates the number of sig- nificant digits, conversion to e (or E) format is performed.	double type	data.
с	c conver- sion	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 conver- sion	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 charac- ters specified by the precision. (Null characters are not output. Space, horizontal tab, and new-line charac- ters are not included in the converted string.)	Pointer to char type	The precision specification indi- cates the number of characters to be output. If the precision is omit- ted, characters are output up to, but not including, the null charac- ter in the string pointed to by the data. (Null characters are not out- put. Space, horizontal tab, and new-line characters are not included in the converted string.)
р	p conver- sion	Assuming data as a pointer, conver- sion is performed to a string of com- piler-defined printable characters.	Pointer to void type	The precision specification is invalid.
n	No conver- sion is per- formed.	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 conver- sion is per- formed.	% 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.

snprintf

Converts data according to a format and outputs it to the specified area.

[Format]

#include <stdio.h>

long snprintf(char *restrict s, size_t n, const char *restrict control

[, arg]...);

[Parameters] s Pointer to storage area to which data is to be output n Number of characters to be output control Pointer to string indicating format arg,... Data to be output according to format [Return values] Number of characters converted [Remarks]

The snprintf function converts and edits parameter arg according to the format-representing string pointed to by control, and outputs the result to the storage area pointed to by s.

A null character is appended at the end of the converted and output string. This null character is not included in the return value (number of characters output). For details of the format specifications, see the description of the fprintf function.

vsnprintf

Converts data according to a format and outputs it to the specified area. [Format] #include <stdarg.h> #include <stdio.h> long vsnprintf(char *restrict s, size t n, const char *restrict control, va list arg) [Parameters] s Pointer to storage area to which data is to be output n Number of characters to be output control Pointer to string indicating format arg Parameter list [Return values] Number of characters converted [Remarks] The **vsnprintf** function is equivalent to **snprintf** with **arg** specified instead of the variable parameters. Initialize arg through the va_start macro before calling the vsnprintf function. The vsnprintf function does not call the va_end macro.

fscanf

Inputs data from a stream input/output file and converts it according to a format. [Format]

#include <stdio.h>
long fscanf (FILE *fp, const char *control[, ptr]...);
[Parameters]
fp File pointer
control Pointer to string indicating format
ptr,... Pointer to storage area that stores input data
[Return values]
Normal: Number of data items successfully input and converted
Abnormal: Input data ends before input data conversion is performed: EOF
[Remarks]



The **fscanf** function inputs data from the stream input/output file indicated by file pointer **fp**, converts and edits it according to the string that represents the format pointed to by **control**, and stores the result in the storage area pointed to by **ptr**.

The format specifications for inputting data are shown below.

Overview of Formats

The string that represents the format is made up of the following three kinds of string.

- Space characters

If a space (' '), horizontal tab ('\t'), or new-line character ('\n') is specified, processing is performed to skip to the next non-white-space character in the input data.

- Ordinary characters

If a character that is neither one of the space characters listed above nor % is specified, one input data character is input. The input character must match a character specified in the string that represents the format.

- Conversion specification

A conversion specification is a string beginning with % that specifies the method of converting the input data and storing it in the area pointed to by the following parameter. The conversion specification format conforms to the following rules:

% [*] [Field width] [Converted data size] Conversion specifier

If there is no pointer to the storage area that stores input data corresponding to the conversion specification in the format, the behavior is not guaranteed. In addition, when a pointer to a storage area that stores input data remains though the format is exhausted, that pointer is ignored.

Description of Conversion Specifications

- * specification

Suppresses storage of the input data in the storage area pointed to by the parameter.

- Field width

The maximum number of characters in the data to be input is specified as a decimal number.

- Converted data size

For d, i, o, u, x, X, e, E, and f conversions (see table 6.12), the size (**short** type, **long** type, **long** type, or **long double** type) of the converted data is specified. In other conversions, this specification is ignored. Table 6.11 shows the types of size specification and their meanings.

Table 6.11 Converted Data Size Specification Types and Meanings

Туре	Meaning
h	For d, i, o, u, x, and X conversions, specifies that the converted data is of short type.
I	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.
II	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.

Conver- sion Speci- fier	Conversion Type	Conversion Method	Data Type Subject to Conversion
d	d conver- sion	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 I (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
0	o conver- sion	An octal string is converted to integer type data.	Integer type
u	u conver- sion	An unsigned decimal string is converted to integer type data.	Integer type
x	x conver- sion	A hexadecimal string is converted to integer type data. There is no difference in meaning between \mathbf{x} conversion and \mathbf{X} conver-	Integer type
х	X conver- sion	sion.	
S	s conver- sion	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 conver- sion	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, spec- ify %1s. If the field width is specified, the number of characters equiva- lent to that specification are read. In this case, therefore, the storage area that stores the converted data needs the specified size.	char type
е	e conver- sion	A string indicating a floating-point number is converted to floating-point type data. There is no difference in meaning between the e conversion	Floating-point type
E	E conver- sion	The input format is a floating-point number that can be represented by the strtod function.	
f	f conversion		
g	g conver- sion		
G	G conver- sion		
р	p conver- sion	A string converted by ${\bf p}$ conversion of the fprintf function is converted to pointer type data.	Pointer to void type
n	No conver- sion is per- formed.	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 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 conver- sion is per- formed.	% is read.	None



If the conversion specifier is a letter not shown in table 6.12, the behavior is not guaranteed. For the other characters, the behavior is implementation-defined.

printf

Converts data according to a format and outputs it to the standard output file (stdout).

[Format]
#include <stdio.h>
long printf (const char *control[, arg]...);
[Parameters]
control Pointer to string indicating format
arg,... Data to be output according to format
[Return values]
Normal: Number of characters converted and output
Abnormal: Negative value
[Remarks]
The printf function converts and edits parameter arg according

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.

vfscanf

Inputs data from a stream input/output file and converts it according to a format. [Format] #include <stdarg.h> #include <stdio.h> long vfscanf(FILE *restrict fp, const char *restrict control, va_list arg) [Parameters] fp File pointer control Pointer to wide string indicating format arg Parameter list [Return values] Normal: Number of data items successfully input and converted Abnormal: Input data ends before input data conversion is performed: EOF [Remarks] The vfscanf function is equivalent to fscanf with arg specified instead of the variable parameter list. Initialize arg through the va_start macro before calling the vfscanf function. The vfscanf function does not call the va_end macro.

scanf

Inputs data from the standard input file (stdin) and converts it according to a format. [Format]

#include <stdio.h>
long scanf (const char *control[, ptr...]);
[Parameters]
control Pointer to string indicating format
ptr.... Pointer to storage area that stores input and converted data
[Return values]
Normal: Number of data items successfully input and converted
Abnormal: EOF
[Remarks]

The **scanf** function inputs data from the standard input file (**stdin**), converts and edits it according to the string that represents the format pointed to by **control**, and stores the result in the storage area pointed to by **ptr**.

The **scanf** function returns the number of data items successfully input and converted as the return value. **EOF** is returned if the standard input file ends before the first conversion.

For details of the format specifications, see the description of the **fscanf** function.

For %e conversion, specify I for double type, and specify L for long double type. The default type is float.



vscanf

Inputs data from the specified storage area and converts it according to a format. [Format] #include <stdarg.h> #include <stdarg.h> long vscanf(const char *restrict control, va_list arg) [Parameters] control Pointer to string indicating format arg Parameter list [Return values] Normal: Number of data items successfully input and converted Abnormal: Input data ends before input data conversion is performed: EOF [Remarks] The vscanf function is equivalent to scanf with arg specified instead of the variable parameters. Initialize arg through the va_start macro before calling the vscanf function. The vscanf function does not call the va_end macro.

sprintf

Converts data according to a format and outputs it to the specified area. [Format] #include <stdio.h> long sprintf (char *s, const char *control[, arg...]); [Parameters] s Pointer to storage area to which data is to be output control Pointer to string indicating format

arg,... Data to be output according to format

[Return values]

Number of characters converted

[Remarks]

The **sprintf** function converts and edits parameter **arg** according to the string that represents the format pointed to by **control**, and outputs the result to the storage area pointed to by **s**.

A null character is appended at the end of the converted and output string. This null character is not included in the return value (number of characters output).

For details of the format specifications, see the description of the **fprintf** function.

sscanf		

Inputs data from the specified storage area and converts it according to a format. [Format] #include <stdio.h> long sscanf (const char *s, const char *control[, ptr...]); [Parameters] s Storage area containing data to be input control Pointer to string indicating format ptr,... Pointer to storage area that stores input and converted data [Return values] Normal: Number of data items successfully input and converted Abnormal: EOF [Remarks] The sscanf function inputs data from the storage area pointed to by s, converts and

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.



vsscanf

Inputs data from the specified storage area and converts it according to a format. [Format] #include <stdarg.h> #include <stdio.h> long vsscanf(const char *restrict s, const char *restrict control, va list arg) [Parameters] s Storage area containing data to be input control Pointer to string indicating format arg Parameter list [Return values] Normal: Number of data items successfully input and converted Abnormal: nput data ends before input data conversion is performed: EOF [Remarks] The vsscanf function is equivalent to sscanf with arg specified instead of the variable parameters. Initialize arg through the va_start macro before calling the vsscanf function. The vsscanf function does not call the va_end macro.

vfprintf

Outputs a variable parameter list to the specified stream input/output file according to a format.

[Format] #include <stdarg.h> #include <stdio.h> long vfprintf (FILE *fp, const char *control, va_list arg) [Parameters] fp File pointer control Pointer to string indicating format arg Parameter list [Return values] Normal: Number of characters converted and output Abnormal: Negative value [Remarks]

The **vfprintf** function sequentially converts and edits a variable parameter list according to the string that represents the format pointed to by **control**, and outputs the result to the stream input/output file indicated by **fp**.

The **vfprintf** function returns the number of data items converted and output, or a negative value when an error occurs. Within the **vfprintf** function, the **va_end** macro is not invoked.

For details of the format specifications, see the description of the **fprintf** function.

Parameter **arg**, indicating the parameter list, must be initialized beforehand by the **va_start** macro (and the succeeding **va_arg** macro).

vprintf

Outputs a variable parameter list to the standard output file (stdout) according to a format. [Format] #include <stdarg.h> #include <stdio.h> long vprintf (const char *control, va_list arg) [Parameters] control Pointer to string indicating format arg Parameter list [Return values] Normal: Number of characters converted and output Abnormal: Negative value [Remarks] The **vprintf** function sequentially converts and edits a variable parameter list according to the string that represents the format pointed to by **control**, and outputs the result to the standard output file.

The **vprintf** function returns the number of data items converted and output, or a negative value when an error occurs.



Within the **vprintf** function, the **va_end** macro is not invoked.

For details of the format specifications, see the description of the **fprintf** function.

Parameter **arg**, indicating the parameter list, must be initialized beforehand by the **va_start** macro (and the succeeding **va_arg** macro).

vsprintf

Outputs a variable parameter list to the specified storage area according to a format. [Format] #include <stdarg.h> #include <stdio.h> long vsprintf (char *s, const char *control, va_list arg) [Parameters] s Pointer to storage area to which data is to be output control Pointer to string indicating format arg Parameter list [Return values] Normal: Number of characters converted Abnormal: Negative value [Remarks]

The **vsprintf** function sequentially converts and edits a variable parameter list according to the string that represents the format pointed to by **control**, and outputs the result to the storage area pointed to by **s**.

A null character is appended at the end of the converted and output string. This null character is not included in the return value (number of characters output).

For details of the format specifications, see the description of the **fprintf** function.

Parameter **arg**, indicating the parameter list, must be initialized beforehand by the **va_start** macro (and the succeeding **va_arg** macro).

fgetc

Inputs one character from a stream input/output file. [Format] #include <stdio.h> long fgetc (FILE *fp); [Parameters] fp File pointer [Return values] Normal: End-of-file: EOF Otherwise: Input character Abnormal: EOF [Remarks] When a read error occurs, the error indicator for that file is set. The fgetc function inputs one character from the stream input/output file indicated by file pointer fp.

The **fgetc** function normally returns the input character, but returns **EOF** at end-of-file or when an error occurs. At end-of-file, the end-of-file indicator for that file is set.

fgets

Inputs a string from a stream input/output file. [Format] #include <stdio.h> char *fgets (char *s, long n, FILE *fp); [Parameters] s Pointer to storage area to which string is input n Number of bytes of storage area to which string is input fp File pointer [Return values] Normal: End-of-file: NULL Otherwise: s Abnormal: NULL



[Remarks]

The **fgets** function inputs a string from the stream input/output file indicated by file pointer **fp** to the storage area pointed to by **s**.

The **fgets** function performs input up to the (n–1)th character or a new-line character, or until end-of-file, and appends a null character at the end of the input string.

The **fgets** function normally returns **s**, the pointer to the storage area to which the string is input, but returns **NULL** at end-of-file or if an error occurs.

The contents of the storage area pointed to by **s** do not change at end-of-file, but are not guaranteed when an error occurs.

fputc

Outputs one character to a stream input/output file. [Format] #include <stdio.h> long fputc (long c, FILE *fp); [Parameters] c Character to be output fp File pointer [Return values] Normal: Output character Abnormal: EOF [Remarks] When a write error occurs, the error indicator for that file is set. The **fputc** function outputs character **c** to the stream input/output file indicated by file pointer **fp**. The **fputc** function normally returns **c**, the output character, but returns EOF when an error occurs.

fputs

Outputs a string to a stream input/output file. [Format] #include <stdio.h> long fputs (const char *s, FILE *fp); [Parameters] s Pointer to string to be output fp File pointer [Return values] Normal: 0 Abnormal: Nonzero [Remarks]

The **fputs** function outputs the string pointed to by **s** up to the character preceding the null character to the stream input/ output file indicated by file pointer **fp**. The null character indicating the end of the string is not output. The **fputs** function normally returns zero, but returns nonzero when an error occurs.

getc

Inputs one character from a stream input/output file. [Format] #include <stdio.h> long getc (FILE *fp); [Parameters] fp File pointer [Return values] Normal: End-of-file: EOF Otherwise: Input character Abnormal: EOF [Remarks] When a read error occurs, the error indicator for that file is set. The getc function inputs one character from the stream input/output file indicated by file pointer fp.



The **getc** function normally returns the input character, but returns **EOF** at end-of-file or when an error occurs. At end-of-file, the end-of-file indicator for that file is set.

getchar

Inputs one character from the standard input file (stdin). [Format] #include <stdio.h> long getchar (void); [Return values] Normal: End-of-file: EOF Otherwise: Input character Abnormal: EOF [Remarks] When a read error occurs, the error indicator for that file is set. The getchar function inputs one character from the standard input file (stdin). The getchar function normally returns the input character, but returns EOF at end-of-file or when an error occurs. At end-of-file, the end-of-file indicator for that file is set.

gets

Inputs a string from the standard input file (stdin). [Format] #include <stdio.h> char *gets (char *s); [Parameters] s Pointer to storage area to which string is input [Return values] Normal: End-of-file: NULL Otherwise: s Abnormal: NULL [Remarks] The gets function inputs a string from the standard input file (stdin)

The gets function inputs a string from the standard input file (stdin) to the storage area starting at s.

The **gets** function inputs characters up to end-of-file or until a new-line character is input, and appends a null character instead of a new-line character.

The **gets** function normally returns **s**, the pointer to the storage area to which the string is input, but returns **NULL** at the end of the standard input file or when an error occurs.

The contents of the storage area pointed to by **s** do not change at the end of the standard input file, but are not guaranteed when an error occurs.

putc

Outputs one character to a stream input/output file. [Format] #include <stdio.h> long putc (long c, FILE *fp); [Parameters] c Character to be output fp File pointer [Return values] Normal: Output character Abnormal: EOF [Remarks] When a write error occurs, the error indicator for that file is set. The putc function outputs character c to the stream input/output file indicated by file pointer fp. The putc function normally returns c, the output character, but returns EOF when an error occurs.



putchar

Outputs one character to the standard output file (stdout). [Format] #include <stdio.h> long putchar (long c); [Parameters] c Character to be output [Return values] Normal: Output character Abnormal: EOF [Remarks] When a write error occurs, the error indicator for that file is set. The putchar function outputs character c to the standard output file (stdout). The putchar function normally returns c, the output character, but returns EOF when an error occurs.

puts

Outputs a string to the standard output file (stdout). [Format] #include <stdio.h> long puts (const char *s); [Parameters] s Pointer to string to be output [Return values] Normal: 0 Abnormal: Nonzero [Remarks] The puts function outputs the string pointed to by s to the standard output file (stdout). The p

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.

ungete

Returns one character to a stream input/output file. [Format] #include <stdio.h>

long ungetc (long c, FILE *fp);
[Parameters]
c Character to be returned
fp File pointer
[Return values]
Normal: Returned character
Abnormal: EOF
[Remarks]

The **ungetc** function returns character **c** to the stream input/output file indicated by file pointer **fp**. Unless the **fflush**, **fseek**, or **rewind** function is called, this returned character will be the next input data.

The **ungetc** function normally returns **c**, which is the returned character, but returns **EOF** when an error occurs. The behavior is not guaranteed when the **ungetc** function is called more than once without intervening **fflush**, **fseek**, or **rewind** function execution. When the **ungetc** function is executed, the current file position indicator for that file is moved back one position; however, when this file position indicator has already been positioned at the beginning of the file, its value is not guaranteed.

fread

Inputs data from a stream input/output file to the specified storage area.
[Format]
#include <stdio.h>
size_t fread (void *ptr, size_t size, size_t n, FILE *fp);



[Parameters]

ptr Pointer to storage area to which data is input size Number of bytes in one member n Number of members to be input fp File pointer [Return values] When **size** or **n** is 0: 0 When **size** and **n** are both nonzero: Number of successfully input members [Remarks]

The **fread** function inputs **n** members whose size is specified by **size**, from the stream input/output file indicated by file pointer **fp**, into the storage area pointed to by **ptr**. The file position indicator for the file is advanced by the number of bytes input.

The **fread** function returns the number of members successfully input, which is normally the same as the value of **n**. However, at end-of-file or when an error occurs, the number of members successfully input so far is returned, and then the return value will be less than **n**. The **ferror** and **feof** functions should be used to distinguish between end-of-file and error occurrence.

When the value of **size** or **n** is zero, zero is returned as the return value and the contents of the storage area pointed to by **ptr** do not change. When an error occurs or when only a part of the members can be input, the file position indicator is not guaranteed.

fwrite

Outputs data from a memory area to a stream input/output file.

[Format]

#include <stdio.h>

size_t fwrite (const void *ptr, size_t size, size_t n, FILE *fp);

[Parameters]

ptr Pointer to storage area storing data to be output

size Number of bytes in one member

n Number of members to be output

fp File pointer

[Return values]

Number of successfully output members

[Remarks]

The **fwrite** function outputs **n** members whose size is specified by **size**, from the storage area pointed to by **ptr**, to the stream input/output file indicated by file pointer **fp**. The file position indicator for the file is advanced by the number of bytes output.

The **fwrite** function returns the number of members successfully output, which is normally the same as the value of \mathbf{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 \mathbf{n} .

When an error occurs, the file position indicator is not guaranteed.

fseek

Shifts the current read/write position in a stream input/output file.

[Format]
#include <stdio.h>
long fseek (FILE *fp, long offset, long type);
[Parameters]
fp File pointer
offset Offset from position specified by type of offset
type Type of offset
[Return values]
Normal: 0
Abnormal: Nonzero
[Remarks]
The fseek function shifts the current read/write position in the s

The **fseek** function shifts the current read/write position in the stream input/output file indicated by file pointer **fp** by **off-set** bytes from the position specified by **type** (the type of offset).

The types of offset are shown in table 6.13.

The fseek function normally returns zero, but returns nonzero in response to an invalid request.



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.

Table 6.13 Types of Offset

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.

Obtains the current read/write position in a stream input/output file.

```
[Format]
#include <stdio.h>
long ftell (FILE *fp);
[Parameters]
fp File pointer
[Return values]
Current file position indicator position (text file)
Number of bytes from beginning of file to current position (binary file)
```

[Remarks]

ftell

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.

rowind	
rewind	
1011110	

Shifts the current read/write position in a stream input/output file to the beginning of the file.

```
[Format]
#include <stdio.h>
void rewind (FILE *fp);
[Parameters]
fp File pointer
[Remarks]
```

The **rewind** function shifts the current read/write position in the stream input/output file indicated by file pointer **fp**, to the beginning of the file.

The **rewind** function clears the end-of-file indicator and error indicator for the file. Note that calling the **rewind** function cancels the effect of the **ungetc** function.

clearerr

Clears the error state of a stream input/output file. [Format] #include <stdio.h> void clearerr (FILE *fp); [Parameters] fp File pointer [Remarks]

The **clearerr** function clears the error indicator and end-of-file indicator for the stream input/output file indicated by file pointer **fp**.



feof

Tests for the end of a stream input/output file.

```
[Format]
#include <stdio.h>
long feof (FILE *fp);
[Parameters]
fp File pointer
[Return values]
End-of-file: Nonzero
Otherwise: 0
[Remarks]
The feof function tests for the end of the stream in:
```

The feof function tests for the end of the stream input/output file indicated by file pointer fp.

The **feof** function tests the end-of-file indicator for the specified stream input/output file, and if the indicator is set, returns nonzero to indicate that the file is at its end. If the end-of-file indicator is not set, the **feof** function returns zero to show that the file is not yet at its end.

ferror

Tests for stream input/output file error state. [Format] #include <stdio.h> long ferror (FILE *fp); [Parameters] fp File pointer [Return values] If file is in error state: Nonzero Otherwise: 0 [Remarks]

The ferror function tests whether the stream input/output file indicated by file pointer fp is in the error state.

The **ferror** function tests the error indicator for the specified stream input/output file, and if the indicator is set, returns nonzero to show that the file is in the error state. If the error indicator is not set, the **ferror** function returns zero to show that the file is not in the error state.

Outputs an error message corresponding to the error number to the standard error file (stderr).

[Format] #include <stdio.h>

void perror (const char *s)

[Parameters]

s Pointer to error message

[Remarks]

The **perror** function maps **errno** to the error message indicated by **s**, and outputs the message to the standard error file (**stderr**).

If **s** is not **NULL** and the string pointed to by **s** is not a null character, the output format is as follows: the string pointed to by **s** followed by a colon and space, then the implementation-defined error message, and finally a new-line character.

6.4.12 <stdlib.h>

Defines standard functions for standard processing of C programs. The following macros are implementation-defined.



Туре	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 Idiv function.
	lldiv_t	Indicates the type of structure of the value returned by the Ildiv function.
Constant (macro)	RAND_MAX	Indicates the maximum value of pseudo-random integers generated by the rand function.
	EXIT_SUCCESS	Indicates the successfully completed state.
Function	atof	Converts a number-representing string to a double type floating-point number.
	atoi	Converts a decimal-representing string to an int type integer.
	atol	Converts a decimal-representing string to a long type integer.
	atoll	Converts a decimal-representing string to a long long type integer.
	strtod	Converts a number-representing string to a double type floating-point number.
	strtof	Converts a number-representing string to a float type floating-point number.
	strtold	Converts a number-representing string to a long double type floating-point number.
	strtol	Converts a number-representing string to a long type integer.
	strtoul	Converts a number-representing string to an unsigned long type integer.
	strtoll	Converts a number-representing string to a long long type integer.
	strtoull	Converts a number-representing string to an unsigned long long type integer.
	rand	Generates pseudo-random integers from 0 to RAND_MAX .
	srand	Sets an initial value of the pseudo-random number sequence generated by the rand function.
	calloc	Allocates a storage area and clears all bits in the allocated storage area to 0.
	free	Releases specified storage area.
	malloc	Allocates a storage area.
	realloc	Changes the size of storage area to a specified value.
	bsearch	Performs binary search.
	qsort	Performs sorting.
	abs	Calculates the absolute value of an int type integer.



Туре	Definition Name	Description
Function	div	Carries out division of int type integers and obtains the quotient and remainder.
	labs	Calculates the absolute value of a long type integer.
	ldiv	Carries out division of long type integers and obtains the quotient and remainder.
	llabs	Calculates the absolute value of a long long type integer.
	lldiv	Carries out division of long long type integers and obtains the quotient and remainder.
	mbstowcs	Converts a multibyte string to a wide string.
	wcstombs	Converts a wide string to a multibyte string.

Implementation-Defined Specifications

Item	Compiler Specifications
calloc , malloc , or realloc function operation when the size is 0.	NULL is returned.

atof

Converts a number-representing string to a double type floating-point number.

[Format]
#include <stdlib.h>
double atof (const char *nptr);
[Parameters]
nptr Pointer to a number-representing string to be converted
[Return values]
Converted data as a double type floating-point number
[Remarks]
If the converted result overflows or underflows, errno is set.
Data is converted up to the first character that does not fit the floating-point data type.
The atof function does not guarantee the return value if an error such as an overflow occurs. When you want to acquire
the guaranteed return value, use the strtod function.

atoi

Converts a decimal-representing string to an int type integer. [Format] #include <stdlib.h> long atoi (const char *nptr); [Parameters] nptr Pointer to a number-representing string to be converted [Return values] Converted data as an int type integer [Remarks] If the converted result overflows, **errno** is set. Data is converted up to the first character that does not fit the decimal data type.

The **atoi** function does not guarantee the return value if an error such as an overflow occurs. When you want to acquire the guaranteed return value, use the **strtol** function.

atol

Converts a decimal-representing string to a long type integer. [Format]



#include <stdlib.h>
long atol (const char *nptr);
[Parameters]
nptr Pointer to a number-representing string to be converted
[Return values]
Converted data as a long type integer
[Remarks]
If the converted result overflows, errno is set.
Data is converted up to the first character that does not fit the decimal data type.

The **atol** function does not guarantee the return value if an error such as an overflow occurs. When you want to acquire the guaranteed return value, use the **strtol** function.

atoll			

Converts a decimal-representing string to a **long long** type integer. [Format] #include <stdlib.h> long long atoll (const char *nptr); [Parameters] nptr Pointer to a number-representing string to be converted [Return values] Converted data as a **long long** type integer [Remarks] If the converted result overflows, **errno** is set.

Data is converted up to the first character that does not fit the decimal data type.

The **atoll** function does not guarantee the return value if an error such as an overflow occurs. When you want to acquire the guaranteed return value, use the **strtoll** function.

strtod

Converts a number-representing string to a double type floating-point number.

[Format]

#include <stdlib.h>
double strtod (const char *nptr, char **endptr);

[Parameters]

nptr Pointer to a number-representing string to be converted

endptr Pointer to the storage area containing a pointer to the first character that does not represent a floating-point number

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent a floating-point number: 0 If the string pointed by **nptr** begins with a character that represents a floating-point number: Converted data as a **double** type floating-point number

Abnormal: If the converted data overflows: **HUGE_VAL** with the same sign as that of the string before conversion If the converted data underflows: 0

[Remarks]

The **strtod** function converts data, from the first digit or the decimal point up to the character immediately before the character that does not represent a floating-point number, into a **double** type floating-point number. However, if neither an exponent nor a decimal point is found in the data to be converted, the compiler assumes that the decimal point comes next to the last digit in the string. In the area pointed by **endptr**, the function sets up a pointer to the first character that does not represent a floating-point number. If some characters that do not represent a floating-point number come before digits, the value of **nptr** is set. If **endptr** is **NULL**, nothing is set in this area.

strtof

Converts a number-representing string to a **float** type floating-point number. [Format] #include <stdlib.h> float strtof (const char *nptr, char **endptr); [Parameters] nptr Pointer to a number-representing string to be converted



endptr Pointer to the storage area containing a pointer to the first character that does not represent a floating-point number

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent a floating-point number: 0 If the string pointed by **nptr** begins with a character that represents a floating-point number: Converted data as a **float** type floating-point number

Abnormal: If the converted data overflows: **HUGE_VALF** with the same sign as that of the string before conversion If the converted data underflows: 0

[Remarks]

If the converted result overflows or underflows, errno is set.

The **strtof** function converts data, from the first digit or the decimal point up to the character immediately before the character that does not represent a floating-point number, into a **float** type floating-point number. However, if neither an exponent nor a decimal point is found in the data to be converted, the compiler assumes that the decimal point comes next to the last digit in the string. In the area pointed by **endptr**, the function sets up a pointer to the first character that does not represent a floating-point number. If some characters that do not represent a floating-point number come before digits, the value of **nptr** is set. If **endptr** is **NULL**, nothing is set in this area.

strtold

Converts a number-representing string to a long double type floating-point number.

[Format]

#include <stdlib.h>

long double strtold (const char *nptr, char **endptr);

[Parameters]

nptr Pointer to a number-representing string to be converted

endptr Pointer to the storage area containing a pointer to the first character that does not represent a floating-point number

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent a floating-point number: 0 If the string pointed by **nptr** begins with a character that represents a floating-point number: Converted data as a **long double** type floating-point number

Abnormal: If the converted data overflows: **HUGE_VALL** with the same sign as that of the string before conversion If the converted data underflows: 0

[Remarks]

If the converted result overflows or underflows, errno is set.

The **strtold** function converts data, from the first digit or the decimal point up to the character immediately before the character that does not represent a floating-point number, into a **long double** type floating-point number. However, if neither an exponent nor a decimal point is found in the data to be converted, the compiler assumes that the decimal point comes next to the last digit in the string. In the area pointed by **endptr**, the function sets up a pointer to the first character that does not represent a floating-point number. If some characters that do not represent a floating-point number come before digits, the value of **nptr** is set. If **endptr** is **NULL**, nothing is set in this area.

strtol

Converts a number-representing string to a long type integer.

[Format]

#include <stdlib.h>

long strtol (const char *nptr, char **endptr, long base);

[Parameters]

nptr Pointer to a number-representing string to be converted

endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer

base Radix of conversion (0 or 2 to 36)

[Return values]

Normal: If the string pointed by nptr begins with a character that does not represent an integer: 0

If the string pointed by nptr begins with a character that represents an integer: Converted data as a long type integer

Abnormal: If the converted data overflows: LONG_MAX or LONG_MIN depending on the sign of the string before conversion

[Remarks]

If the converted result overflows, errno is set.

The **strtol** function converts data, from the first digit up to the character before the first character that does not represent an integer, into a **long** type integer.

In the storage area pointed by **endptr**, the function sets up a pointer to the first character that does not represent an integer. If some characters that do not represent an integer come before the first digit, the value of **nptr** is set in this area. If **endptr** is **NULL**, nothing is set in this area.

If the value of **base** is 0, the rules described in section 3.1.3 (4), Integers, are observed at conversion. If the value of **base** is 2 to 36, it indicates the radix of conversion, where a (or A) to z (or Z) in the string to be converted correspond to numbers 10 to 35. If a character that is not smaller than the **base** value is found in the string to be converted, conversion stops immediately. A 0 after a sign is ignored at conversion. Similarly, 0x (or 0X) is ignored when **base** is 16.

strtoul

Converts a number-representing string to an unsigned long type integer. [Format] #include <stdlib.h> unsigned long strtoul (const char *nptr, char **endptr, long base); [Parameters] nptr Pointer to a number-representing string to be converted endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer base Radix of conversion (0 or 2 to 36) [Return values] Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0 If the string pointed by **nptr** begins with a character that represents an integer: Converted data as an **unsigned long** type integer Abnormal: If the converted data overflows: **ULONG_MAX**

[Remarks]

If the converted result overflows, errno is set.

The **strtoul** function converts data, from the first digit up to the character before the first character that does not represent an integer, into an **unsigned long** type integer.

In the storage area pointed by **endptr**, the function sets up a pointer to the first character that does not represent an integer. If some characters that do not represent an integer come before the first digit, the value of **nptr** is set in this area. If **endptr** is **NULL**, nothing is set in this area.

If the value of **base** is 0, the rules described in section 3.1.3 (4), Integers, are observed at conversion. If the value of **base** is 2 to 36, it indicates the radix of conversion, where a (or A) to z (or Z) in the string to be converted correspond to numbers 10 to 35. If a character that is not smaller than the **base** value is found in the string to be converted, conversion stops immediately. A 0 after a sign is ignored at conversion. Similarly, 0x (or 0X) is ignored when **base** is 16.

Converts a number-representing string to a long long type integer.

[Format]

#include <stdlib.h>

long long strtoll (const char *nptr, char **endptr, long base);

[Parameters]

nptr Pointer to a number-representing string to be converted

endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer base Radix of conversion (0 or 2 to 36)

[Return values]

Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0

If the string pointed by **nptr** begins with a character that represents an integer: Converted data as a **long long** type integer Abnormal: If the converted data overflows: **LLONG_MAX** or **LLONG_MIN** depending on the sign of the string before

conversion [Remarks]

If the converted result overflows, **errno** is set.

The **strtoll** function converts data, from the first digit up to the character before the first character that does not represent an integer, into a **long long** type integer.

In the storage area pointed by **endptr**, the function sets up a pointer to the first character that does not represent an integer. If some characters that do not represent an integer come before the first digit, the value of **nptr** is set in this area. If **endptr** is **NULL**, nothing is set in this area.

If the value of **base** is 0, the rules described in section 3.1.3 (4), Integers, are observed at conversion. If the value of **base** is 2 to 36, it indicates the radix of conversion, where a (or A) to z (or Z) in the string to be converted correspond to numbers 10 to 35. If a character that is not smaller than the **base** value is found in the string to be converted, conversion stops immediately. A 0 after a sign is ignored at conversion. Similarly, 0x (or 0X) is ignored when **base** is 16.

strtoull

Converts a number-representing string to an unsigned long long type integer. [Format] #include <stdlib.h> unsigned long long strtoull (const char *nptr, char **endptr, long base); [Parameters] nptr Pointer to a number-representing string to be converted endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer base Radix of conversion (0 or 2 to 36) [Return values] Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0 If the string pointed by **nptr** begins with a character that represents an integer: Converted data as an **unsigned long long** type integer Abnormal: If the converted data overflows: **ULLONG_MAX** [Remarks] If the converted result overflows error is set

If the converted result overflows, **errno** is set.

The **strtoull** function converts data, from the first digit up to the character before the first character that does not represent an integer, into an **unsigned long long** type integer.

In the storage area pointed by **endptr**, the function sets up a pointer to the first character that does not represent an integer. If some characters that do not represent an integer come before the first digit, the value of **nptr** is set in this area. If **endptr** is **NULL**, nothing is set in this area.

If the value of **base** is 0, the rules described in section 3.1.3 (4), Integers, are observed at conversion. If the value of **base** is 2 to 36, it indicates the radix of conversion, where a (or A) to z (or Z) in the string to be converted correspond to numbers 10 to 35. If a character that is not smaller than the **base** value is found in the string to be converted, conversion stops immediately. A 0 after a sign is ignored at conversion. Similarly, 0x (or 0X) is ignored when **base** is 16.

rand

Generates a pseudo-random integer from 0 to RAND_MAX. [Format] #include <stdlib.h> long rand (void); [Return values] Pseudo-random integer

srand

Sets an initial value of the pseudo-random number sequence generated by the rand function.

[Format]
#include <stdlib.h>
void srand (unsigned long seed);
[Parameters]
seed Initial value for pseudo-random number sequence generation
[Remarks]
The srand function sets up an initial value for pseudo-random number sequence generation of the rand function. If

pseudo-random number sequence generation by the **rand** function is repeated and if the same initial value is set up again by the **srand** function, the same pseudo-random number sequence is repeated.

If the **rand** function is called before the **srand** function, 1 is set as the initial value for the pseudo-random number generation.

calloc

Allocates a storage area and clears all bits in the allocated storage area to 0. [Format] #include <stdlib.h> void *calloc (size_t nelem, size_t elsize); [Parameters] nelem Number of elements



elsize Number of bytes occupied by a single element [Return values] Normal: Starting address of an allocated storage area Abnormal: Storage allocation failed, or either of the parameter is 0: **NULL** [Remarks]

The **calloc** function allocates as many storage units of size **elsize** (bytes) as the number specified by nelem. The function also clears all the bits in the allocated storage area to 0.

free

Releases the specified storage area. [Format] #include <stdlib.h> void free (void *ptr); [Parameters] ptr Address of storage area to release [Remarks]

The **free** function releases the storage area pointed by **ptr**, to enable reallocation for use. If **ptr** is **NULL**, the function carries out nothing.

If the storage area attempted to release was not allocated by the **calloc**, **malloc**, or **realloc** function, or when the area has already been released by the **free** or **realloc** function, correct operation is not guaranteed. Operation result of reference to a released storage area is also not guaranteed.

```
malloc
```

Allocates a storage area. [Format] #include <stdlib.h> void *malloc (size_t size); [Parameters] size Size in number of bytes of storage area to allocate [Return values] Normal: Starting address of allocated storage area Abnormal: Storage allocation failed, or **size** is 0: **NULL** [Remarks] The **malloc** function allocates a storage area of a specified number of bytes by **size**.

realloc	
---------	--

Changes the size of a storage area to a specified value. [Format] #include <stdlib.h> void *realloc (void *ptr, size_t size); [Parameters] ptr Starting address of storage area to be changed size Size of storage area in number of bytes after the change [Return values] Normal: Starting address of storage area whose size has been changed Abnormal: Storage area allocation has failed, or size is 0: NULL [Remarks] The **realloc** function changes the size of the storage area specified by **p**

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.

bsearch

Performs binary search.



[Format]

#include <stdlib.h>

void *bsearch (const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *)); [Parameters] key Pointer to data to find base Pointer to a table to be searched nmemb Number of members to be searched size Number of bytes of a member to be searched compar Pointer to a function that performs comparison [Return values] If a matching member is found: Pointer to the matching member If no matching member is found: NULL [Remarks] The bsearch function searches the table specified by base for a member that matches the data specified by key, by

The **bsearch** function searches the table specified by **base** for a member that matches the data specified by **key**, by binary search method. The function that performs comparison should receive pointers **p1** (first parameter) and **p2** (second parameter) to two data items to compare, and return the result complying with the specification below.

*p1 < *p2: Returns a negative value.

*p1 == *p2: Returns 0.

*p1 > *p2: Returns a positive value.

Members to be searched must be placed in the ascending order.

qsort

Performs sorting.
[Format]
#include <stdlib.h>
void qsort (const void *base, size_t nmemb, size_t size, int (*compar)(const void *,
const void *));
[Parameters]
base Pointer to the table to be sorted
nmemb Number of members to sort
size Number of bytes of a member to be sorted
compar Pointer to a function to perform comparison
[Remarks]
The **qsort** function sorts out data on the table pointed to by **base**. The data arrangement order is specified by the

pointer to a function to perform comparison. This comparison function should receive pointers **p1** (first parameter) and **p2** (second parameter) as two data items to be compared, and return the result complying with the specification below.

*p1 < *p2: Returns a negative value.

*p1 == *p2: Returns 0.

*p1 > *p2: Returns a positive value.

abs

Calculates the absolute value of an int type integer. [Format] #include <stdlib.h> long abs (long i); [Parameters] i Integer to calculate the absolute value [Return values] Absolute value of i [Remarks] If the resultant absolute value cannot be expressed as an **int** type integer, correct operation is not guaranteed.

div

Carries out division of **int** type integers and obtains the quotient and remainder. [Format]



#include <stdlib.h>
div_t div (long numer, long denom);
[Parameters]
numer Dividend
denom Divisor
[Return values]
Quotient and remainder of division of numer by denom

labs

Calculates the absolute value of a **long** type integer. [Format] #include <stdlib.h> long labs (long j); [Parameters] j Integer to calculate the absolute value [Return values] Absolute value of j [Remarks] If the resultant absolute value cannot be expressed as a **long** type integer, correct operation is not guaranteed.

ldiv

Carries out division of long type integers and obtains the quotient and remainder.

[Format]
#include <stdlib.h>
ldiv_t ldiv (long numer, long denom);
[Parameters]
numer Dividend
denom Divisor
[Return values]
Quotient and remainder of division of numer by denom

llabs

Calculates the absolute value of a **long long** type integer. [Format] #include <stdlib.h> long long llabs (long long j); [Parameters] j Integer to calculate the absolute value [Return values] Absolute value of j [Remarks] If the resultant absolute value cannot be expressed as a **long long** type integer, correct operation is not guaranteed.

lldiv

Carries out division of **long long** type integers and obtains the quotient and remainder. [Format] #include <stdlib.h> Ildiv_t lldiv (long long numer, long long denom); [Parameters] numer Dividend denom Divisor [Return values] Quotient and remainder of division of **numer** by **denom**



mbstowcs

Converts a multibyte string to a wide string.

[Format]

#include <stdlib.h>
size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);
[Parameters]
pwcs Pointer to wide string
s Pointer to multibyte string
n Number of wide characters to be stored in wide string
[Return values]
Normal: Number of characters written to wide string
Abnormal: (size_t)(-1): An illegal multibyte sequence is detected.
[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.

```
wcstombs
```

Converts a wide string to a multibyte string.

[Format]
#include <stdlib.h>
size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);
[Parameters]
s Pointer to multibyte string
pwcs Pointer to wide string
n Number of bytes to be written to multibyte string
[Return values]
Normal: Number of bytes written to multibyte string
Abnormal: (size_t)(-1): An illegal multibyte sequence is detected
[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 \mathbf{n} , the array is not terminated by a null character.



6.4.13 <string.h>

Defines functions for handling character arrays.

Туре	Definition Name	Description
Function	тетсру	Copies contents of a source storage area of a specified length to a destination storage area.
strcpystrncpystrcatstrncatmemcmpstrcmpstrcmpstrncmpmemchrstrchrstrcspnstrpbrk	strcpy	Copies contents of a source string including the null character to a destination storage area.
	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.
	Searches a specified storage area for the first occurrence of a specified charac- ter.	
	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 consec- utive characters at the beginning that are not included in another string speci- fied.
	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 strstr strtok memset strerror strlen	strspn	Checks a specified string from the beginning and counts the number of consec- utive 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.
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**.



(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.



Subsequent data is copied in succession.

```
memcpy
```

Copies the contents of a source storage area of a specified length to a destination storage area. [Format] #include <string.h> void *memcpy (void *s1, const void *s2, size_t n); [Parameters] s1 Pointer to destination storage area s2 Pointer to source storage area n Number of characters to be copied [Return values] s1 value

strcpy

Copies the contents of a source string including the null character to a destination storage area.
[Format]
#include <string.h>
char *strcpy (char *s1, const char *s2);
[Parameters]



s1 Pointer to destination storage areas2 Pointer to source string[Return values]s1 value

strncpy

Copies a source string of a specified length to a destination storage area. [Format] #include <string.h> char *strncpy (char *s1, const char *s2, size_t n); [Parameters] s1 Pointer to destination storage area s2 Pointer to source string n Number of characters to be copied [Return values] s1 value [Remarks] The strncpy function copies up to n characters from the beginning of the

The **strncpy** function copies up to **n** characters from the beginning of the string pointed by **s2** to a storage area pointed by **s1**. If the length of the string specified by **s2** is shorter than **n** characters, the function elongates the string to the length by padding with null characters.

If the length of the string specified by **s2** is longer than **n** characters, the copied string in **s1** storage area ends with a character other than the null character.

strcat

Concatenates a string after another string. [Format] #include <string.h> char *strcat (char *s1, const char *s2); [Parameters] s1 Pointer to the string after which another string is appended s2 Pointer to the string to be appended after the other string [Return values] s1 value [Remarks] The strcat function concatenates the string specified by s2 at the end of another string specified by s1. The null charac-

strncat

Concatenates a string of a specified length after another string. [Format] #include <string.h> char *strncat (char *s1, const char *s2, size_t n); [Parameters] s1 Pointer to the string after which another string is appended s2 Pointer to the string to be appended after the other string n Number of characters to concatenate [Return values] s1 value [Remarks]

The **strncat** function concatenates up to **n** characters from the beginning of the string specified by **s2** at the end of another string specified by **s1**. The null character at the end of the **s1** string is replaced by the first character of the **s2** string. A null character is appended to the end of the concatenated string.

ter indicating the end of the s2 string is also copied. The null character at the end of the s1 string is deleted.

memcmp

Compares the contents of two storage areas specified.



[Format]

#include <string.h>
long memcmp (const void *s1, const void *s2, size_t n);
[Parameters]
s1 Pointer to the reference storage area to be compared
s2 Pointer to the storage area to compare to the reference
n Number of characters to compare
[Return values]
If storage area pointed by s1 > storage area pointed by s2: Positive value
If storage area pointed by s1 < storage area pointed by s2: 0
If storage area pointed by s1 < storage area pointed by s2: Negative value
[Remarks]</pre>

The **memcmp** function compares the contents of the first **n** characters in the storage areas pointed by **s1** and **s2**. The rules of comparison are implementation-defined.

strcmp

Compares the contents of two strings specified.

[Format]
#include <string.h>
long strcmp (const char *s1, const char *s2);
[Return values]
If string pointed by s1 > string pointed by s2: Positive value
If string pointed by s1 < string pointed by s2: 0
If string pointed by s1 < string pointed by s2: Negative value
[Parameters]
s1 Pointer to the reference string to be compared
s2 Pointer to the string to compare to the reference</pre>

[Remarks]

The **strcmp** function compares the contents of the strings pointed by **s1** and **s2**, and sets up the comparison result as a return value. The rules of comparison are implementation-defined.

strncmp

Compares two strings specified up to a specified length. [Format] #include <string.h> long strncmp (const char *s1, const char *s2, size_t n); [Parameters] s1 Pointer to the reference string to be compared s2 Pointer to the reference string to be compared s2 Pointer to the string to compare to the reference n Maximum number of characters to compare [Return values] If string pointed by s1 > string pointed by s2: Positive value If string pointed by s1 == string pointed by s2: 0 If string pointed by s1 < string pointed by s2: Negative value [Remarks]

The **strncmp** function compares the contents of the strings pointed by **s1** and **s2**, up to **n** characters. The rules of comparison are implementation-defined.

memchr

Searches a specified storage area for the first occurrence of a specified character.
[Format]
#include <string.h>
void *memchr (const void *s, long c, size_t n);
[Parameters]
s Pointer to the storage area to be searched
c Character to search for
n Number of characters to search


[Return values]

If the character is found: Pointer to the found character If the character is not found: **NULL** [Remarks]

The **memchr** function searches the storage area specified by **s** from the beginning up to **n** characters, looking for the first occurrence of the character specified as **c**. If the **c** character is found, the function returns the pointer to the found character.

strchr

Searches a specified string for the first occurrence of a specified character.

[Format]

char *strchr (const char *s, long c); [Parameters] s Pointer to the string to be searched c Character to search for [Return values] If the character is found: Pointer to the found character If the character is not found: NULL [Remarks]

The **strchr** function searches the string specified by **s** looking for the first occurrence of the character specified as **c**. If the **c** character is found, the function returns the pointer to the found character.

The null character at the end of the s string is included in the search object.

strcspn

Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are not included in another string specified.

[Format]

```
#include <string.h>
size_t strcspn (const char *s1, const char *s2);
[Parameters]
s1 Pointer to the string to be checked
s2 Pointer to the string used to check s1
[Return values]
Number of characters at the beginning of the s1 string that are not included in the s2 string
[Remarks]
The strcspn function checks from the beginning of the string specified by s1, counts the nu
```

The **strcspn** function checks from the beginning of the string specified by **s1**, counts the number of consecutive characters that are not included in another string specified by **s2**, and returns that length.

The null character at the end of the s2 string is not taken as a part of the s2 string.

strpbrk

Searches a specified string for the first occurrence of the character that is included in another string specified. [Format]

#include <string.h>
char *strpbrk (const char *s1, const char *s2);
[Parameters]
s1 Pointer to the string to be searched
s2 Pointer to the string that indicates the characters to search s1 for
[Return values]
If the character is found: Pointer to the found character
If the character is not found: NULL
[Remarks]
The strapt function searches the string specified by s1 looking for the

The **strpbrk** function searches the string specified by **s1** looking for the first occurrence of any character included in the string specified by **s2**. If any searched character is found, the function returns the pointer to the first occurrence.



strrchr

```
Searches a specified string for the last occurrence of a specified character.

[Format]

#include <string.h>

char *strrchr (const char *s, long c);

[Parameters]

s Pointer to the string to be searched

c Character to search for

[Return values]

If the character is found: Pointer to the found character

If the character is not found: NULL

[Remarks]

The attracter function searches the string specified by a looking for the last of
```

The **strrchr** function searches the string specified by **s** looking for the last occurrence of the character specified by **c**. If the **c** character is found, the function returns the pointer to the last occurrence of that character.

The null character at the end of the **s** string is included in the search objective.

strspn

Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are included in another string specified.

[Format]
#include <string.h>
size_t strspn (const char *s1, const char *s2);
[Parameters]
s1 Pointer to the string to be checked
s2 Pointer to the string used to check s1
[Return values]
Number of characters at the beginning of the s1 string that are included in the s2 string
[Remarks]
The strspn function checks from the beginning of the string specified by s1, counts the

The **strspn** function checks from the beginning of the string specified by **s1**, counts the number of consecutive characters that are included in another string specified by **s2**, and returns that length.

strstr

Searches a specified string for the first occurrence of another string specified.

```
[Format]
#include <string.h>
char *strstr (const char *s1, const char *s2);
[Parameters]
s1 Pointer to the string to be searched
s2 Pointer to the string to search for
[Return values]
If the string is found: Pointer to the found string
If the string is not found: NULL
[Remarks]
The strstr function searches the string specified by s1 looking for
```

The strstr function searches the string specified by s1 looking for the first occurrence of another string specified by s2, and returns the pointer to the first occurrence.

strtok

```
Divides a specified string into some tokens.

[Format]

#include <string.h>

char *strtok (char *s1, const char *s2);

[Return values]

If division into tokens is successful: Pointer to the first token divided

If division into tokens is unsuccessful: NULL

[Parameters]
```



s1 Pointer to the string to be divided into some tokens

s2 Pointer to the string representing string-dividing characters

[Remarks]

The **strtok** function should be repeatedly called to divide a string.

(a) First call

The string pointed by **s1** is divided at a character included in the string pointed by **s2**. If a token has been separated, the function returns a pointer to the beginning of that token. Otherwise, the function returns **NULL**.

(b) Second and subsequent calls

Starting from the next character separated before as the token, the function repeats division at a character included in the string pointed by **s2**. If a token has been separated, the function returns a pointer to the beginning of that token. Otherwise, the function returns **NULL**.

At the second and subsequent calls, specify **NULL** as the first parameter. The string pointed by **s2** can be changed at each call. The null character is appended at the end of a separated token.

An example of use of the **strtok** function is shown below.

Example

```
1 #include <string.h>
2 static char s1[ ]="a@b, @c/@d";
3 char *ret;
4
5 ret = strtok(s1, "@");
6 ret = strtok(NULL, ",@");
7 ret = strtok(NULL, "/@");
8 ret = strtok(NULL, "@");
```

Explanation:

The above example program uses the **strtok** function to divide string "a@b, @c/@d" into tokens a, b, c, and d. The second line specifies string "a@b, @c/@d" as an initial value for string **s1**.

The fifth line calls the **strtok** function to divide tokens using '@' as the delimiter. As a result, a pointer to character 'a' is returned, and the null character is embedded at '@,' the first delimiter after character 'a.' Thus string 'a' has been separated.

Specify **NULL** for the first parameter to consecutively separate tokens from the same string, and repeat calling the **strtok** function.

Consequently, the function separates strings 'b,' 'c,' and 'd.'

memset

Sets a specified character a specified number of times at the beginning of a specified storage area.

```
[Format]
#include <string.h>
void *memset (void *s, long c, size_t n);
[Parameters]
s Pointer to storage area to set characters in
c Character to be set
n Number of characters to be set
[Return values]
Value of s
[Remarks]
The memset function sets the character specified by c a number of times specified by n in the storage area specified by
```

S.

strerror

Returns an error message corresponding to a specified error number. [Format] #include <string.h> char *strerror (long s); [Parameters] s Error number [Return values] Pointer to the error message (string) corresponding to the specified error number [Remarks]



The strerror function receives an error number specified by **s** and returns an error message corresponding to the number. Contents of error messages are implementation-defined.

If the returned error message is modified, correct operation is not guaranteed.

strlen

```
Calculates the length of a string.

[Format]

#include <string.h>

size_t strlen (const char *s);

[Parameters]

s Pointer to the string to check the length of

[Return values]

Number of characters in the string

[Remarks]

The null character at the end of the s string is excluded from the string length.
```

memmove

Copies the specified size of the contents of a source area to a destination storage area. If part of the source storage area and the destination storage area overlap, data is copied to the destination storage area before the overlapped source storage area is overwritten. Therefore, correct copy is enabled.

[Format]
#include <string.h>
void *memmove (void *s1, const void *s2, size_t n);
[Parameters]
s1 Pointer to the destination storage area
s2 Pointer to the source storage area
n Number of characters to be copied
[Return values]
Value of s1



6.4.14 <complex.h>

Performs various complex number operations. For **double**-type complex number functions, the definition names are used as function names without change. For **float**-type and **long double**-type function names, "f" and "l" are added to the end of definition names, respectively.

Туре	Definition Name	Description
Function	CACOS <-lang=c99>	Calculates the arc cosine of a complex number.
	casin <-lang=c99>	Calculates the arc sine of a complex number.
	catan <-lang=c99>	Calculates the arc tangent of a complex number.
	CCOS <-lang=c99>	Calculates the cosine of a complex number.
	CSin <-lang=c99>	Calculates the sine of a complex number.
	ctan <-lang=c99>	Calculates the tangent of a complex number.
	cacosh <-lang=c99>	Calculates the arc hyperbolic cosine of a complex number.
	casinh <-lang=c99>	Calculates the arc hyperbolic sine of a complex number.
	catanh <-lang=c99>	Calculates the arc hyperbolic tangent of a complex number.
	ccosh <-lang=c99>	Calculates the hyperbolic cosine of a complex number.
	csinh <-lang=c99>	Calculates the hyperbolic sine of a complex number.
	ctanh <-lang=c99>	Calculates the hyperbolic tangent of a complex number.
	<pre>Cexp <-lang=c99></pre>	Calculates the natural logarithm base e raised to the complex power 2.
	clog <-lang=c99>	Calculates the natural logarithm of a complex number.
	cabs <-lang=c99>	Calculates the absolute value of a complex number.
	CPOW <-lang=c99>	Calculates a power of a complex number.
	csqrt <-lang=c99>	Calculates the square root of a complex number.
	Carg <-lang=c99>	Calculates the argument of a complex number.
	cimag <-lang=c99>	Calculates the imaginary part of a complex number.
	CONj <-lang=c99>	Reverses the sign of the imaginary part and calculates the complex conjugate of a complex number.
	cproj <-lang=c99>	Calculates the projection of a complex number on Riemann sphere.
	creal <-lang=c99>	Calculates the real part of a complex number.

cacosf/cacos/cacosl

Calculates the arc cosine of a complex number. [Format] #include <complex.h> float complex cacosf(float complex z); double complex cacos(double complex z); long double complex cacosl(long double complex z); [Parameters] z Complex number for which arc cosine is to be computed [Return values] Normal:Complex arc cosine of z Abnormal: Domain error: Returns not-a-number. [Remarks]



A domain error occurs for a value of **z** not in the range [-1.0, 1.0].

The **cacos** function returns the arc cosine in the range $[0, \pi]$ on the real axis and in the infinite range on the imaginary axis.

casinf/casin/casinl

```
Calculates the arc sine of a complex number.

[Format]

#include <complex.h>

float complex casinf(float complex z);

double complex casin(double complex z);

long double complex casinl(long double complex z);

[Parameters]

z Complex number for which arc sine is to be computed

[Return values]

Normal: Complex arc sine of z

Abnormal: Domain error: Returns not-a-number.

[Remarks]

A domain error occurs for a value of z not in the range [-1.0, 1.0].

The casin function returns the arc sine in the range [-\pi/2, \pi/2] on the real axis and in the infinite range on the imaginary

axis.
```

catanf/catan/catanl

Calculates the arc tangent of a complex number. [Format]

```
#include <complex.h>
float complex catanf(float complex z);
double complex catan(double complex z);
long double complex catanl(long double complex z);
[Parameters]
z Complex number for which arc tangent is to be computed
[Return values]
Normal: Complex arc tangent of z
[Remarks]
The catan function returns the arc tangent in the range [-π/2 π/2] on
```

The **catan** function returns the arc tangent in the range $[-\pi/2, \pi/2]$ on the real axis and in the infinite range on the imaginary axis.

ccosf/ccos/ccosl

```
Calculates the cosine of a complex number.

[Format]

#include <complex.h>

float complex ccosf(float complex z);

double complex ccos(double complex z);

long double complex ccosl(long double complex z);

[Parameters]

z Complex number for which cosine is to be computed

[Return values]

Complex cosine of z
```

csinf/csin/csinl

Calculates the sine of a complex number.
[Format]
#include <complex.h>
float complex csinf(float complex z);
double complex csin(double complex z);
long double complex csinl(long double complex z);



[Parameters] z Complex number for which sine is to be computed [Return values] Complex sine of **z**

ctanf/ctan/ctanl

Calculates the tangent of a complex number. [Format] #include <complex.h> float complex ctanf(float complex z); double complex ctan(double complex z); long double complex ctanl(long double complex z); [Parameters] z Complex number for which tangent is to be computed [Return values] Complex tangent of z

cacoshf/cacosh/cacoshl

Calculates the arc hyperbolic cosine of a complex number. [Format]

#include <complex.h>
float complex cacoshf(float complex z);
double complex cacosh(double complex z);
long double complex cacoshl(long double complex z);
[Parameters]
z Complex number for which arc hyperbolic cosine is to be computed
[Return values]
Normal: Complex arc hyperbolic cosine of z
Abnormal: Domain error: Returns not-a-number.
[Remarks]
A domain error occurs for a value of z not in the range [-1.0, 1.0].
The cacoshf function returns the arc hyperbolic cosine in the range [0, π].

casinhf/casinh/casinhl

Calculates the arc hyperbolic sine of a complex number. [Format] #include <complex.h> float complex casinhf(float complex z); double complex casinh(double complex z); long double complex casinhl(long double complex z); [Parameters] z Complex number for which arc hyperbolic sine is to be computed [Return values] Complex arc hyperbolic sine of z

catanhf/catanh/catanhl





Complex arc hyperbolic tangent of z

ccoshf/ccosh/ccoshl

Calculates the hyperbolic cosine of a complex number. [Format]

#include <complex.h>
float complex ccosh(float complex z);
double complex ccosh(double complex z);
long double complex ccoshl(long double complex z);
[Parameters]
z Complex number for which hyperbolic cosine is to be computed
[Return values]
Complex hyperbolic cosine of z

csinhf/csinh/csinhl

Calculates the hyperbolic sine of a complex number. [Format] #include <complex.h> float complex csinhf(float complex z); double complex csinh(double complex z); long double complex csinhl(long double complex z); [Parameters] z Complex number for which hyperbolic sine is to be computed [Return values] Complex hyperbolic sine of z

ctanhf/ctanh/ctanhl

Calculates the hyperbolic tangent of a complex number. [Format] #include <complex.h> float complex ctanhf(float complex z); double complex ctanh(double complex z); long double complex ctanhl(long double complex z); [Parameters] z Complex number for which hyperbolic tangent is to be computed [Return values] Complex hyperbolic tangent of z

cexpf/cexp/cexpl

Calculates the exponential function value of a complex number. [Format] #include <complex.h> float complex cexpf(float complex z); double complex cexp(double complex z); long double complex cexpl(long double complex z); [Parameters] z Complex number for which exponential function is to be computed [Return values] Exponential function value of z

clogf/clog/clogl

Calculates the natural logarithm of a complex number. [Format]



#include <complex.h>
float complex clogf(float complex z);
double complex clog(double complex z);
long double complex clogl(long double complex z);
[Parameters]
z Complex number for which natural logarithm is to be computed
[Return values]
Normal: Natural logarithm of z
Abnormal: Domain error: Returns not-a-number.
[Remarks]
A domain error occurs if z is negative.
A range error occurs if z is 0.0.
The clog function returns the natural logarithm in the infinite range on the real axis and in the range [-iπ, +iπ] on the

cabsf/cabs/cabsl

imaginary axis.

Calculates the absolute value of a complex number. [Format] #include <complex.h> float cabsf(float complex z); double cabs(double complex z); long double cabsl(long double complex z); [Return values] Absolute value of z [Parameters] z Complex number for which absolute value is to be computed

cpowf/cpow/cpowl

Calculates a power of a complex number. [Format] #include <complex.h> float complex cpowf(float complex x, float complex y); double complex cpow(double complex x, double complex y); long double complex cpowl(long double complex x, long double complex y); [Parameters] x Value to be raised to a power y Power value [Return values] Normal: Value of x raised to the power y Abnormal: Domain error: Returns not-a-number. [Remarks] A domain error occurs if x is 0.0 and y is 0.0 or smaller, or if x is negative and y is not an integer. The branch cut for the first parameter of the **cpow** function group is along the negative real axis.

csqrtf/csqrt/csqrtl

Calculates the square root of a complex number. [Format] #include <complex.h> float complex csqrtf(float complex z); double complex csqrt(double complex z); long double complex csqrtl(long double complex z); [Parameters] z Complex number for which the square root is to be computed [Return values] Normal: Complex square root of z Abnormal: Domain error: Returns not-a-number. [Remarks]



A domain error occurs if **z** is negative.

The branch cut for the **csqrt** function group is along the negative real axis.

The range of the return value from the csqrt function group is the right halfplane including the imaginary axis.

cargf/carg/cargl

Calculates the argument. [Format] #include <complex.h> float cargf(float complex z); double carg(double complex z); long double cargl(long double complex z); [Parameters] z Complex number for which the argument is to be computed [Return values] Argument value of z [Remarks] The branch cut for the carg function group is along the negative real axis. The carg function group returns the argument in the range [-π, +π].

cimagf/cimag/cimagl

Calculates the imaginary part. [Format] #include <complex.h> float cimagf(float complex z); double cimag(double complex z); long double cimagl(long double complex z); [Parameters] z Complex number for which the imaginary part is to be computed [Return values] Imaginary part value of z as a real number

conjf/conj/conjl

Reverses the sign of the imaginary part of a complex number and calculates the complex conjugate. [Format]

```
#include <complex.h>
float complex conjf(float complex z);
double complex conj(double complex z);
long double complex conjl(long double complex z);
[Parameters]
z Complex number for which the complex conjugate is to be computed
[Return values]
Complex conjugate of z
```

cprojf/cproj/cprojl

Calculates the projection of a complex number on the Riemann sphere.
[Format]
#include <complex.h>
float complex cprojf(float complex z);
double complex cproj(double complex z);
long double complex cprojl(long double complex z);
[Parameters]
z Complex number for which the projection on the Riemann sphere is to be computed
[Return values]
Projection of z on the Riemann sphere



crealf/creal/creall

```
Calculates the real part of a complex number.

[Format]

#include <complex.h>

float crealf(float complex z);

double creal(double complex z);

long double creall(long double complex z);

[Parameters]

z Complex number for which the real part value is to be computed

[Return values]

Real part value of z
```

6.4.15 <fenv.h>

Provides access to the floating-point environment. The following macros and functions are all implementation-defined.



Туре	Definition Name	Description
Туре	fenv_t	Indicates the type of the entire floating-point environment.
(macro)	fexcept_t	Indicates the type of the floating-point status flags.
Constant (macro)	FE_DIVBYZERO FE_INEXACT FE_INVALID FE_OVERFLOW FE_UNDERFLOW FE_ALL_EXCEPT	Indicates the values (macros) defined when the floating-point exception is supported.
	FE_DOWNWARD FE_TONEAREST FE_TOWARDZERO FE_UPWARD	Indicates the values (macros) of the floating-point rounding direction.
	FE_DFL_ENV	Indicates the default floating-point environment of the program.
Function	feclearexcept	Attempts to clear a floating-point exception.
	fegetexceptflag	Attempts to store the state of a floating-point flag in an object.
	feraiseexcept	Attempts to generate a floating-point exception.
	fesetexceptflag	Attempts to set a floating-point flag.
	fetestexcept	Checks if floating-point flags are set.
	fegetround	Gets the rounding direction.
	fesetround	Sets the rounding direction.
	fegetenv	Attempts to get the floating-point environment.
	feholdexcept	Saves the floating-point environment, clears the floating-point status flags, and sets the non-stop mode for the floating-point exceptions.
	fesetenv	Attempts to set the floating-point environment.
	feupdateenv	Attempts to save the floating-point exceptions in the automatic storage, set the floating-point environment, and generate the saved floating-point exceptions.

feclearexcept

Attempts to clear a floating-point exception. [Format] #include <fenv.h> long feclearexcept(long e); [Parameters] e Floating-point exception [Return values] Normal: 0 Abnormal: Nonzero [Remarks] Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fegetexceptflag

Gets the state of a floating-point flag. [Format]



#include <fenv.h>
long fegetexceptflag(fexcept_t *f, long e);
[Parameters]
f Pointer to area to store the exception flag state
e Value indicating the exception flag whose state is to be acquired
[Return values]
Normal: 0
Abnormal: Nonzero
[Remarks]
Do not use this function when compiler option nofpu is selected. If used

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

feraiseexcept

Attempts to generate a floating-point exception. [Format] #include <fenv.h> long feraiseexcept(long e); [Return values] Normal: 0 Abnormal: Nonzero [Parameters] e Value indicating the exception to be generated [Remarks] When generating an "overflow" or "underflow" floating the exception of the second secon

When generating an "overflow" or "underflow" floating-point exception, whether the **feraiseexcept** function also generates an "inexact" floating-point exception is implementation-defined.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fesetexceptflag

Sets the state of an exception flag. [Format] #include <fenv.h> long fesetexceptflag(const fexcept_t *f, long e); [Parameters] f Pointer to the source location from which the exception flag state is to be acquired e Value indicating the exception flag whose state is to be set [Return values] Normal: 0 Abnormal: Nonzero [Remarks] Before calling the fesetexceptflag function, specify a flag state in the source location through the fegetexceptflag function.

The fesetexceptflag function only sets the flag state without generating the corresponding floating-point exception.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fetestexcept

Checks the exception flag states. [Format] #include <fenv.h> long fetestexcept(long e); [Parameters] e Value indicating flags whose states are to be checked (multiple flags can be specified) [Return values] Bitwise OR of e and floating-point exception macros [Remarks] A single fetestexcept function call can check multiple floating-point exceptions.



Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fegetround

Gets the current rounding direction. [Format]

#include <fenv.h>
 long fegetround(void);
[Return values]
Normal: 0

Abnormal: Negative value when there is no rounding direction macro value or the rounding direction cannot be determined

[Remarks]

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fesetround

Sets the current rounding direction. [Format] #include <fenv.h> #include <assert.h> long fesetround(long rnd); [Return values] 0 only when the rounding direction has been set successfully [Remarks] The rounding direction is not changed if the rounding direction requests through the **fesetround** function differs from the rounding macro value.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fegetenv

Gets the floating-point environment. [Format] #include <fenv.h> long fegetenv(fenv_t *f); [Parameters] f Pointer to area to store the floating-point environment [Return values] Normal: 0 Abnormal: Nonzero [Remarks] Do not use this function when compiler option **petru** is on

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

feholdexcept

Saves the floating-point environment.
[Format]
#include <fenv.h>
long feholdexcept(fenv_t *f);
[Parameters]
f Pointer to the floating-point environment
[Return values]
0 only when the environment has been saved successfully
[Remarks]



When saving the floating-point function environment, the **feholdexcept** function clears the floating-point status flags and sets the non-stop mode for all floating-point exceptions. In non-stop mode, execution continues even after a floatingpoint exception occurs.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

fesetenv

Sets the floating-point environment. [Format] #include <fenv.h> long fesetenv(const fenv_t *f); [Parameters] f Pointer to the floating-point environment [Return values] Normal: 0 Abnormal: Nonzero [Remarks] For the argument of this function, specify the

For the argument of this function, specify the environment stored or saved by the **fegetenv** or **feholdexcept** function, or the environment equal to the floating-point environment macro.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.

feupdateenv

Sets the floating-point environment with the previously generated exceptions retained.

[Format] #include <fenv.h> long feupdateenv(const fenv_t *f); [Parameters] f Pointer to the floating-point environment to be set [Return values] Normal: 0 Abnormal: Nonzero [Remarks]

For the argument of this function, specify the object stored or saved by the **fegetenv** or **feholdexcept** function call, or the floating-point environment equal to the floating-point environment macro.

Do not use this function when compiler option **nofpu** is selected. If used, the function returns a nonzero value, which indicates an abnormality.



6.4.16 <inttypes.h>

Extends the integer types.

The following macros and functions are all implementation-defined.

Туре	Definition Name	Description
Type (macro)	Imaxdiv_t	Indicates the type of the value returned by the imaxdiv function.
Variable (macro)	PRIdN PRIdLEASTN PRIdFASTN PRIdFASTN PRIdPTR PRIiN PRIILEASTN PRIIFASTN PRIIFASTN PRIOLEASTN PRIOEASTN PRIOFASTN PRIOPTR PRIUN PRIULEASTN PRIUFASTN PRIUFASTN PRIUPTR PRIXN PRIXLEASTN PRIXLEASTN PRIXLEASTN PRIXFASTN PRIXFASTN PRIXPTR	



Туре	Definition Name	Description
Variable (macro)	PRIXN PRIXLEASTN PRIXFASTN PRIXMAX PRIXPTR SCNdN SCNdLEASTN SCNdFASTN SCNdFASTN SCNILEASTN SCNILEASTN SCNIFASTN SCNIFASTN SCNOFASTN SCNOFASTN SCNOFASTN SCNOFASTN SCNULEASTN SCNULEASTN SCNULEASTN SCNUFASTN SCNUFASTN SCNUPTR SCNXN SCNXLEASTN SCNXLEASTN SCNXFASTN SCNXFASTN SCNXMAX SCNXPTR	
Function	imaxabs	Calculates the absolute value.
	imaxdiv	Calculates the quotient and remainder.
	strtoimax strtoumax	Equivalent to the strtol , strtol , strtoul , and strtoul functions, except that the initial part of the string is converted to intmax_t and uintmax_t representation.
	wcstoimax wcstoumax	Equivalent to the wcstol , wcstoll , wcstoul , and wcstoull functions except that the initial part of the wide string is converted to intmax_t and uintmax_t representation.

imaxabs

Calculates the absolute value. [Format] #include <inttypes.h> intmax_t imaxabs(intmax_t a); [Parameters]

a Value for which the absolute value is to be computed



[Return values] Absolute value of **a**

imaxdiv

Performs a division operation. [Format] #include <inttypes.h> intmaxdiv_t imaxdiv(intmax_t n, intmax_t d); [Parameters] n Dividend and divisor d [Return values] Division result consisting of the quotient and remainder

strtoimax/strtoumax

Converts a number-representing string to an **intmax_t type** integer.

[Format]

#include <inttypes.h>

intmax_t strtoimax(const char *nptr, char **endptr, long base);

uintmax_t strtoumax(const char *nptr, char **endptr, long base);

[Parameters]

nptr Pointer to a number-representing string to be converted

endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer base Radix of conversion (0 or 2 to 36)

[Return Values]

Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0

If the string pointed by **nptr** begins with a character that represents an integer: Converted data as an **intmax_t** type integer

Abnormal: If the converted data overflows: **INTMAX_MAX**, **INTMAX_MIN**, or **UINTMAX_MAX**

[Remarks] If the converted result overflows, **ERANGE** is set in **errno**.

The strtoimax and strtoumax functions are equivalent to the strtol, strtoll, strtoul, and strtoull functions except that the initial part of the string is respectively converted to intmax_t and uintmax_t integers.

wcstoimax/wcstoumax

Converts a number-representing string to an intmax t or uintmax t type integer. [Format] #include <stddef.h> #include <inttypes.h> intmax t wcstoimax(const wchar t * restrict nptr, wchar t ** restrict endptr, long base); uintmax t wcstoumax(const wchar t * restrict nptr, wchar t ** restrict endptr, long base); [Parameters] nptr Pointer to a number-representing string to be converted endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer base Radix of conversion (0 or 2 to 36) [Return Values] Normal: If the string pointed by **nptr** begins with a character that does not represent an integer: 0 If the string pointed by **nptr** begins with a character that represents an integer: Converted data as an **intmax** t type integer Abnormal: If the converted data overflows: INTMAX_MAX, INTMAX_MIN, or UINTMAX_MAX [Remarks] If the converted result overflows, ERANGE is set in errno. The wcstoimax and wcstoumax functions are equivalent to the wcstol, wcstoll, wcstoul, and wcstoull functions, except that the initial part of the string is respectively converted to intmax_t and uintmax_t integers.



6.4.17 <iso646.h>

This header file defines macros only.

Туре	Definition Name	Description
Macro	and	&&
	and_eq	&=
	bitand	&
	bitor	1
	compl	~
	not	!
	not_eq	!=
	or	П
	or_eq	=
	xor	^
	xor_eq	^=

6.4.18 <stdbool.h>

This header file defines macros only.

Туре	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.



6.4.19 <stdint.h>

This header file defines macros only.

Туре	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 <i>N_</i> MIN INT <i>N_</i> MAX UINT <i>N_</i> MAX	Indicates the minimum value of exact-width signed integer type. Indicates the maximum value of exact-width signed integer type. Indicates the maximum value of exact-width unsigned integer type.
	INT_LEAST <i>N</i> _MIN INT_LEAST <i>N</i> _MAX UINT_LEAST <i>N</i> _MAX	Indicates the minimum value of minimum-width signed integer type. Indicates the maximum value of minimum-width signed integer type. Indicates the maximum value of minimum-width unsigned integer type.
	INT_FAST <i>N</i> _MIN INT_FAST <i>N</i> _MAX UINT_FAST <i>N</i> _MAX	Indicates the minimum value of fastest minimum-width signed integer type. Indicates the maximum value of fastest minimum-width signed integer type. Indicates the maximum value of fastest minimum-width unsigned integer type.
	INTPTR_MIN INTPTR_MAX UINTPTR_MAX	Indicates the minimum value of pointer-holding signed integer type. Indicates the maximum value of pointer-holding signed integer type. Indicates the maximum value of pointer-holding unsigned integer type.
	INTMAX_MIN INTMAX_MAX UINTMAX_MAX	Indicates the minimum value of greatest-width signed integer type. Indicates the maximum value of greatest-width signed integer type. Indicates the maximum value of greatest-width unsigned integer type.
	PTRDIFF_MIN PTRDIFF_MAX	-65535 +65535
	SIG_ATOMIC_MIN SIG_ATOMIC_MAX	-127 +127
	SIZE_MAX	65535



Туре	Definition Name	Description
Macro	WCHAR_MIN WCHAR_MAX	0 65535U
	WINT_MIN WINT_MAX	0 4294967295U
Function (macro)	INT <i>N</i> _C UINT <i>N</i> _C	Expanded to an integer constant expression corresponding to Int_leastN_t. Expanded to an integer constant expression corresponding to Uint_leastN_t.
	INT_MAX_C UINT_MAX_C	Expanded to an integer constant expression with type intmax_t . Expanded to an integer constant expression with type uintmax_t .

6.4.20 <tgmath.h>

This header file defines macros only.

Type-Generic Macro	<math.h> Functions</math.h>	<complex.h> Functions</complex.h>
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
ехр	ехр	сехр
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	_



Type-Generic Macro	<math.h> Functions</math.h>	<complex.h> Functions</complex.h>
expm1	expm1	—
fdim	fdim	_
floor	floor	_
fma	fma	_
fmax	fmax	_
fmin	fmin	_
fmod	fmod	_
frexp	frexp	—
hypot	hypot	_
ilogb	ilogb	_
ldexp	ldexp	_
Igamma	Igamma	_
llrint	llrint	_
llround	llround	_
log10	log10	_
log1p	log1p	_
log2	log2	—
logb	logb	_
Irint	Irint	_
Iround	Iround	—
nearbyint	nearbyint	—
nextafter	nextafter	_
nexttoward	nexttoward	_
remainder	remainder	—
remquo	remquo	_
rint	rint	_
round	round	—
scalbn	scalbn	—
scalbin	scalbin	_
tgamma	tgamma	_
trunc	trunc	—
carg	—	carg
cimag	_	cimag
conj	—	conj
cproj	_	cproj



Type-Generic Macro	<math.h> Functions</math.h>	<complex.h> Functions</complex.h>
creal	_	creal

6.4.21 <wchar.h>

The following shows macros.

Туре	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 spec- ified arguments.
	vwprintf	Equivalent to wprintf with the variable argument list replaced by va_list .
	fwscanf	Inputs and converts data from the stream under control of the wide string and assigns it to an object.
	vfwscanf <-lang=c99>	Equivalent to fwscanf with the variable argument list replaced by va_list .
	swscanf	Converts data under control of the wide string and assigns it to an object.
	vswscanf <-lang=c99>	Equivalent to swscanf with the variable argument list replaced by va_list .
	wscanf	Equivalent to fwscanf with stdin added as an argument before the speci- fied arguments.
	vwscanf <-lang=c99>	Equivalent to wscanf with the variable argument list replaced by va_list .
	fgetwc	Inputs a wide character as the wchar_t type and converts it to the wint_t type.
	fgetws	Stores a sequence of wide characters in an array.
	fputwc	Writes a wide character.



Туре	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 <-1ang=c99> wcstoul wcstoull <-1ang=c99>	These convert the initial part of a wide string to long int , long long int , unsigned long int , or unsigned long long int representation.
	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 ${\bf n}$ 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.



Туре	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 charac- ters 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 ${\bf n}$ 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 corre- sponding wide characters.
	wcsrtombs	Converts a sequence of wide characters to a sequence of corresponding multibyte characters.

fwprintf

Outputs data to a stream input/output file according to the format. [Format] #include <stdio.h> #include <wchar.h> long fwprintf(FILE *restrict fp, const wchar_t *restrict control [, arg] ...); [Parameters] fp File pointer control Pointer to wide string indicating format arg, ... List of data to be output according to format [Return values] Normal: Number of wide strings converted and output Abnormal: Negative value [Remarks] The fwprintf function is the wide-character version of the fprintf function.

vfwprintf

Outputs a variable parameter list to the specified stream input/output file according to a format. [Format] #include <stdarg.h> #include <stdio.h> #include <wchar.h> long vfwprintf(FILE *restrict fp, const char *restrict control, va_list arg) [Parameters] fp File pointer control Pointer to wide string indicating format arg Parameter list



[Return values] Normal: Number of characters converted and output Abnormal:Negative value [Remarks] The **vfwprintf** function is the wide-character version of the **vfprintf** function.

swprintf

Converts data according to a format and outputs it to the specified area. [Format] #include <stdio.h> #include <wchar.h> long swprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict control [, arg]...); [Parameters] s Pointer to storage area to which data is to be output n Number of wide characters to be output control Pointer to wide string indicating format arg,... Data to be output according to format [Return values] Normal: Number of characters converted Abnormal: When a representation format error occurs or writing n or morewide characters is requested: Negative value [Remarks] A representation format error occurs if an illegal multibyte string is passed to the **mbrtowc()** function. The swprintf function is the wide-character version of the sprintf function.

```
vswprintf
```

Outputs a variable parameter list to the specified storage area according to a format. [Format] #include <stdarg.h> #include <swchar.h> long vswprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict control, va_list arg) [Parameters] s Pointer to storage area to which data is to be output n Number of wide characters to be output control Pointer to wide string indicating format arg Parameter list [Return values] Normal: Number of characters converted Abnormal: Negative value [Remarks] The vswprintf function is the wide-character version of the vsprintf function.

wprintf

Converts data according to a format and outputs it to the standard output file (stdout). [Format] #include <stdio.h> #include <wchar.h> long wprintf(const wchar_t *restrict control [, arg]...); [Parameters] control Pointer to string indicating format arg,... Data to be output according to format [Return values] Normal: Number of wide characters converted and output Abnormal:Negative value [Remarks] The wprintf function is the wide-character version of printf function.



vwprintf

Outputs a variable parameter list to the standard output file (**stdout**) according to a format. [Format] #include <stdarg.h> #include <wchar.h> long vwprintf(const wchar_t *restrict control, va_list arg); [Parameters] control Pointer to wide string indicating format arg Parameter list [Return values] Normal: Number of characters converted and output Abnormal:Negative value [Remarks] The **vwprintf** function is the wide-character version of the **vprintf** function.

fwscanf

Inputs data from a stream input/output file and converts it according to a format.
[Format]
#include <stdio.h>
#include <stdio.h>
Iong fwscanf(FILE *restrict fp, const wchar_t *restrict control [, ptr]...);
[Parameters]
fp File pointer
control Pointer to wide string indicating format
ptr Pointer to storage area that stores input data
[Return values]
Normal: Number of data items successfully input and converted
Abnormal: Input data ends before input data conversion is performed: EOF
[Remarks]
The fwscanf function is the wide-character version of the fscanf function.

```
vfwscanf
```

Inputs data from a stream input/output file and converts it according to a format. [Format] #include <stdarg.h> #include <stdarg.h> #include <wchar.h> long vfwscanf(FILE *restrict fp, const wchar_t *restrict control, va_list arg) [Parameters] fp File pointer control Pointer to wide string indicating format arg Parameter list [Return values] Normal: Number of data items successfully input and converted Abnormal: Input data ends before input data conversion is performed: EOF [Remarks] The vfwscanf is the wide-character version of the vfscanf function.

swscanf

Inputs data from the specified storage area and converts it according to a format.
[Format]
#include <stdio.h>
#include <wchar.h>
long swscanf(const wchar_t *restrict s, const wchar_t *restrict control [, ptr]...);
[Parameters]



s Storage area containing data to be input control Pointer to wide string indicating format ptr,... Pointer to storage area that stores input and converted data [Return values] Normal: Number of data items successfully input and converted Abnormal: **EOF** [Remarks] The **swscanf** is the wide-character version of the **sscanf** function.

v3vv3cum

Inputs data from the specified storage area and converts it according to a format.
[Format]
#include <stdarg.h>
#include <wchar.h>
long vswscanf(const wchar_t *restrict s, const wchar_t *restrict control, va_list arg);
[Parameters]
s Storage area containing data to be input
control Pointer to wide string indicating format
arg Parameter list
[Return values]
Normal: Number of data items successfully input and converted
Abnormal: EOF

wscanf

Inputs data from the standard input file (**stdin**) and converts it according to a format. [Format]

#include <wchar.h>
long wscanf(const wchar_t *control [, ptr] ...);
[Parameters]
control Pointer to wide string indicating format
ptr,... Pointer to storage area that stores input and converted data
[Return values]
Normal: Number of data items successfully input and converted
Abnormal: EOF
[Remarks]
The wscanf function is the wide-character version of the scanf function.

vwscanf

Inputs data from the specified storage area and converts it according to a format. [Format] #include <stdarg.h> #include <wchar.h> long vwscanf(const wchar_t *restrict control, va_list arg) [Parameters] control Pointer to wide string indicating format arg Parameter list [Return values] Normal: Number of data items successfully input and converted Abnormal: Input data ends before input data conversion is performed: **EOF** [Remarks] The **vwscanf** function is provided to support wide-character format with the **vscanf** function.

fgetwc

Inputs one wide character from a stream input/output file. [Format]



#include <stdio.h>
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *fp);
[Parameters]
fp File pointer
[Return values]
Normal: End-of-file: EOF
Otherwise: Input wide character
Abnormal: EOF
[Remarks]
When a read error occurs, the error indicator for that file is set.
The fgetwc function is provided to support wide-character input to the fgetc function.

fgetws

Inputs a wide string from a stream input/output file.
[Format]
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t *restrict s, long n, FILE *fp);
[Parameters]
s Pointer to storage area to which wide string is input
n Number of bytes of storage area to which wide string is input
fp File pointer
[Return values]
Normal: End-of-file: NULL
Otherwise: s
Abnormal: NULL
[Remarks]
The fgetws function is provided to support wide-character input to the fgets function.

fputwc

Outputs one wide character to a stream input/output file. [Format] #include <stdio.h> #include <wchar.h> wint_t fputwc(wchar_t c, FILE *fp); [Parameters] c Character to be output fp File pointer [Return values] Normal: Output wide character Abnormal: **EOF** [Remarks] When a write error occurs, the error indicator for that file is set. The **fputwc** function is the wide-character version of the **fputc** function.

fputws

Outputs a wide string to a stream input/output file.
[Format]
#include <stdio.h>
#include <wchar.h>
long fputws(const wchar_t *restrict s, FILE *restrict fp);
[Parameters]
s Pointer to wide string to be output
fp File pointer
[Return values]
Normal: 0



Abnormal: **EOF** [Remarks]

The **fputws** function is the wide-character version of the **fputs** function.

fwide

Specifies the input unit of a file.
[Format]
#include <stdio.h>
#include <stdio.h>
"#include <wchar.h>
long fwide(FILE *fp, long mode);
[Parameters]
fp File pointer
mode Value indicating the input unit
[Return values]
A wide character is specified as the unit: Value greater than 0
A byte is specified as the unit: Value smaller than 0
No input/output unit is specified: 0
[Remarks]
The fwide function does not change the stream input/output unit that has already been determined.

getwc

Inputs one wide character from a stream input/output file. [Format] #include <stdio.h> #include <stdio.h> #include <wchar.h> long getwc(FILE *fp); [Parameters] fp File pointer [Return values] Normal: End-of-file: WEOF Otherwise: Input wide character Abnormal: EOF [Remarks] When a read error occurs, the error indicator for that file is set. The getwc function is equivalent to fgetwc, but getwc may evaluate fp two or more times because it is implemented as a macro. Accordingly, specify an expression without side effects for fp.

getwchar

Inputs one wide character from the standard input file (stdin). [Format] #include <wchar.h> long getwchar(void); [Return values] Normal: End-of-file: WEOF Otherwise: Input wide character Abnormal: EOF [Remarks] When a read error occurs, the error indicator for that file is set. The getwchar function is the wide-character version of the getchar function.

putwc

Outputs one wide character to a stream input/output file. [Format]



#include <stdio.h> #include <wchar.h> wint_t putwc(wchar_t c, FILE *fp); [Parameters] c Wide character to be output fp File pointer [Return values] Normal: Output wide character Abnormal: WEOF [Remarks] When a write error occurs, the error indicator for that file is set.

The putwc function is equivalent to fputwc, but putwc may evaluate fp two or more times because it is implemented as a macro. Accordingly, specify an expression without side effects for fp.

putwchar

Outputs one wide character to the standard output file (stdout). [Format] #include <wchar.h> wint_t putwchar(wchar_t c); [Parameters] c Wide character to be output [Return values] Normal: Output wide character Abnormal: WEOF [Remarks] When a write error occurs, the error indicator for that file is set. The putwchar function is the wide-character version of the putchar function.

ungetwc

Returns one wide character to a stream input/output file. [Format] #include <stdio.h> #include <wchar.h> wint_t ungetwc(wint_t c, FILE *fp); [Parameters] c Wide character to be returned fp File pointer [Return values] Normal: Returned wide character Abnormal: WEOF [Remarks] The ungetwc function is the wide-character version of the ungetc function.

wcstod/wcstof/wcstold

Converts the initial part of a wide string to a specified-type floating-point number.

[Format] #include <wchar.h> double wcstod(const wchar_t *restrict nptr, wchar_t *restrict endptr); float wcstof(const wchar_t *restrict nptr, wchar_t *restrict endptr); long double wcstold(const wchar_t *restrict nptr, wchar_t **restrict endptr); [Parameters] nptr Pointer to a number-representing string to be converted endptr Pointer to the storage area containing a pointer to the first character that does not represent a floating-point number

[Return Values]

Normal: 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 specifiedtype floating-point number

Abnormal: If the converted data overflows: HUGE_VAL, HUGE_VALF, or HUGE_VALL with the same sign as that of the string before conversion

If the converted data underflows: 0

[Remarks]

If the converted result overflows or underflows, errno is set.

The wcstod function group is the wide-character version of the strtod function group.

wcstol/wcstoll/wcstoul/wcstoull

Converts the initial part of a wide string to a specified-type integer.

[Format]

#include <wchar.h>

long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base); long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);

unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);

unsigned long long int wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base;

[Parameters]

nptr Pointer to a number-representing string to be converted

endptr Pointer to the storage area containing a pointer to the first character that does not represent an integer base Radix of conversion (0 or 2 to 36)

[Return values]

Normal: 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

[Remarks]

If the converted result overflows, errno is set.

The wcstol function group is the wide-character version of the strtol function group.

```
wcscpy
```

Copies the contents of a source wide string including the null character to a destination storage area.

[Format]
#include <wchar.h>
wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
[Parameters]
s1 Pointer to destination storage area
s2 Pointer to source string
[Return values]
s1 value
[Remarks]
The wcscpy function group is the wide-character version of the strcpy function group.

wcsncpy

Copies a source wide string of a specified length to a destination storage area. [Format] #include <wchar.h> wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n); [Parameters] s1 Pointer to destination storage area s2 Pointer to source string n Number of characters to be copied [Return values]



s1 value [Remarks]

The **wcsncpy** function is the wide-character version of the **strncpy** function.

wmemcpy

Copies the contents of a source storage area of a specified length to a destination storage area. [Format] #include <wchar.h> wchar_t *wmemcpy(wchar_t *restrict sl, const wchar_t *restrict s2, size_t n); [Parameters] s1 Pointer to destination storage area s2 Pointer to source storage area n Number of characters to be copied [Return values] s1 value [Remarks] The wmemcpy function is the wide-character version of the memcpy function.

wmemmove

Copies the specified size of the contents of a source area to a destination storage area. If part of the source storage area and the destination storage area overlap, data is copied to the destination storage area before the overlapped source storage area is overwritten. Therefore, correct copy is enabled.

[Format]
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
[Parameters]
s1 Pointer to destination storage area
s2 Pointer to source storage area
n Number of characters to be copied
[Return values]
s1 value
[Remarks]
The wmemmove function is the wide-character version of the memmove function.

```
wcscat
```

Concatenates a string after another string. [Format] #include <wchar.h> wchar_t *wcscat(wchar_t *s1, const wchar_t *s2); [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 [Remarks] The **wcscat** function is the wide-character version of the **strcat** function.

wcsncat

Concatenates a string of a specified length after another string. [Format] #include <wchar.h> wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n); [Parameters] s1 Pointer to the string after which another string is appended s2 Pointer to the string to be appended after the other string



n Number of characters to concatenate
[Return values]
s1 value
[Remarks]
The wcsncat function is the wide-character version of the strncat function.

wcscmp

Compares the contents of two strings specified. [Format] #include <wchar.h> long wcscmp(const wchar_t *s1, const wchar_t *s2); [Parameters] s1 Pointer to the reference string to be compared s2 Pointer to the reference string to be compared s2 Pointer to the string to compare to the reference [Return values] If string pointed by s1 > string pointed by s2: Positive value If string pointed by s1 == string pointed by s2: 0 If string pointed by s1 < string pointed by s2: Negative value [Remarks] The wcscmp function is the wide-character version of the strcmp function.

wcsncmp

Compares two strings specified up to a specified length.

[Format]
#include <wchar.h>
long wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
[Parameters]
s1 Pointer to the reference string to be compared
s2 Pointer to the string to compare to the reference
n Maximum number of characters to compare
[Return values]
If string pointed by s1 > string pointed by s2: Positive value
If string pointed by s1 < string pointed by s2: 0
If string pointed by s1 < string pointed by s2: Negative value
[Remarks]
The wcsncmp function is the wide-character version of the strncmp function.</pre>

wmemcmp

Compares the contents of two storage areas specified. [Format] #include <wchar.h> long wmemcmp(const wchar_t * s1, const wchar_t * s2, size_t n); [Parameters] s1 Pointer to the reference storage area to be compared s2 Pointer to the storage area to compare to the reference n Number of characters to compare [Return values] If storage area pointed by s1 > storage area pointed by s2: Positive value If storage area pointed by s1 == storage area pointed by s2: 0 If storage area pointed by s1 < storage area pointed by s2: Negative value [Remarks] The wmemcmp function is the wide-character version of the memcmp function.

wcschr

Searches a specified string for the first occurrence of a specified character.



[Format] #include <wchar.h> wchar_t *wcschr(const wchar_t *s, wchar_t c); [Parameters] s Pointer to the string to be searched c Character to search for [Return values] If the character is found: Pointer to the found character If the character is not found: NULL [Remarks] The wcschr function is the wide-character version of the strchr function.

wcscspn

Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are not included in another string specified.

[Format]

```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
[Parameters]
s1 Pointer to the string to be checked
s2 Pointer to the string used to check s1
[Return values]
Number of characters at the beginning of the s1 string that are not included in the s2 string
[Remarks]
```

The wcscspn function is the wide-character version of the strcspn function.

wcspbrk

Searches a specified string for the first occurrence of the character that is included in another string specified. [Format]

#include <wchar.h> wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2); [Parameters] s1 Pointer to the string to be searched s2 Pointer to the string that indicates the characters to search s1 for [Return values] If the character is found: Pointer to the found character If the character is not found: NULL [Remarks] The wcspbrk function is the wide-character version of the strpbrk function.

wcsrchr

Searches a specified string for the last occurrence of a specified character. [Format] #include <wchar.h> wchar_t *wcsrchr(const wchar_t *s, wchar_t c); [Parameters] s Pointer to the string to be searched c Character to search for [Return values] If the character is found: Pointer to the found character If the character is not found: NULL

wcsspn

Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are included in another string specified.



[Format]

#include <wchar.h>
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
[Parameters]
s1 Pointer to the string to be checked
s2 Pointer to the string used to check s1
[Return values]
Number of characters at the beginning of the s1 string that are included in the s2 string
[Remarks]

The wcsspn function is the wide-character version of the strspn function.

wcsstr

Searches a specified string for the first occurrence of another string specified.

[Format]
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
[Parameters]
s1 Pointer to the string to be searched
s2 Pointer to the string to search for
[Return values]
If the string is found: Pointer to the found string
If the string is not found: NULL

wcstok

Divides a specified string into some tokens.

[Format]

#include <wchar.h>

wchar_t* wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t ** restrict ptr);

[Parameters]

s1 Pointer to the string to be divided into some tokens

s2 Pointer to the string representing string-dividing characters

ptr Pointer to the string where search is to be started at the next function call

[Return values]

If division into tokens is successful: Pointer to the first token divided

If division into tokens is unsuccessful: NULL

[Remarks]

The wcstok function is the wide-character version of the strtok function.

To search the same string for the second or later time, set **s1** to **NULL** and **ptr** to the value returned by the previous function call to the same string.

wmemchr

Searches a specified storage area for the first occurrence of a specified character. [Format] #include <wchar.h> wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n); [Parameters] s Pointer to the storage area to be searched c Character to search for n Number of characters to search [Return values] If the character is found: Pointer to the found character If the character is not found: NULL [Remarks] The wmemchr function is the wide character version of the memchr function

The wmemchr function is the wide-character version of the memchr function.


wcslen

Calculates the length of a wide string except the terminating null wide character. [Format] #include <wchar.h> size_t wcslen(const wchar_t *s); [Parameters] s Pointer to the wide string to check the length of [Return values] Number of characters in the wide string [Remarks] The wcslen function is the wide-character version of the strlen function.

wmemset

Sets a specified character a specified number of times at the beginning of a specified storage area.

```
[Format]
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
[Parameters]
s Pointer to storage area to set characters in
c Character to be set
n Number of characters to be set
[Return values]
Value of s
[Remarks]
The wmemset function is the wide-character version of the memset function.
```

wctob

Converts a wide character to 1-byte representation. [Format] #include <stdio.h> #include <wchar.h> long wctob(wint_t c); [Parameters] c Wide character [Return values] Normal: One-byte value converted from a wide character Abnormal: EOF [Remarks] If the wide character cannot be represented in one byte, EOF is returned. 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.

mbsinit

Checks if a specified **mbstate_t** object indicates the initial conversion state. [Format] #include <wchar.h> long mbsinit(const mbstate_t *ps); [Parameters] ps Pointer to **mbstate_t** object [Return values] Initial conversion state: Nonzero Otherwise: 0



mbrlen

Calculates the number of bytes in a specified multibyte character. [Format] #include <wchar.h> size_t mbrlen(const char * restrict s, size_t n, mbstate_t *restrict ps); [Parameters] s Pointer to multibyte string n Maximum number of bytes to be checked for multibyte character ps Pointer to **mbstate_t** object [Return values] 0: A null wide character is detected in **n** or fewer bytes. From 1 to n inclusive: A multibyte character is detected in **n** or fewer bytes. (size_t)(-2): No complete multibyte character is detected. (size_t)(-1): An illegal multibyte sequence is detected.

mbrtowc

Converts a multibyte character to a wide character.

[Format]

#include <wchar.h>

size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t *
restrict ps);
[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

[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.

[Remarks]

If an illegal multibyte sequence is detected, the **EILSEQ** macro value is set in **errno** and the conversion state is unspecified.

wcrtomb

Converts a wide character to a multibyte character.

[Format]
#include <wchar.h>
size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);
[Parameters]
s Pointer to multibyte string
wc Wide character to be converted
ps Pointer to mbstate_t object
[Return values]
Normal: Number of bytes in the multibyte character
Abnormal: (size_t)(-1): An illegal multibyte sequence is detected
[Remarks]
If an illegal multibyte sequence is detected, the EILSEQ macro value is set in errno and the conversion state is unspecified.

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.



6.5 EC++ Class Libraries

This section describes the specifications of the EC++ class libraries, which can be used as standard libraries in C++ programs. The class library types and corresponding standard include files are described. The specifications of each class library are given in accordance with the library configuration.

- Library types

Table 6.14 shows the class library types and the corresponding standard include files.

Table 6.14 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></iomanip></iostream></ostream></istream></streambuf></ios>
Memory management library	Performs memory allocation and deallo- cation	<new></new>
Complex number calculation class library	Performs calculation of complex num- ber data	<complex></complex>
String manipulation class library	Performs string manipulation	<string></string>

6.5.1 Stream Input/Output Class Library

The header files for stream input/output class libraries are as follows:

- <ios>

Defines data members and function members that specify input/output formats and manage the input/output states. The **<ios>** header file also defines the **Init** and **ios_base** classes in addition to the ios class.

- <streambuf>

Defines functions for the stream buffer.

- <istream> Defines input functions from the input stream.
- <ostream>
 Defines output functions to the output stream.
- <iostream>
 Defines input/output functions.
- <iomanip>
 Defines manipulators with parameters.



The following shows the inheritance relation of the above classes. An arrow (->) indicates that a derived class references a base class. The **streambuf** class has no inheritance relation.



The following types are used by stream input/output class libraries.

Туре	Definition Name	Description
Туре	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

Туре	Definition Name	Description
Variable	init_cnt	Static data member that counts the number of stream input/output objects. The data must be initialized to 0 by a low-level interface.
Function	Init()	Constructor
	~Init()	Destructor

ios_base::Init::Init()

Constructor of class **Init**. Increments **init_cnt**.

ios_base::Init::~Init()

Destructor of class **Init**. Decrements **init_cnt**.

(b) ios_base Class

Туре	Definition Name	Description
Туре	fmtflags	Type that indicates the format control information
	iostate	Type that indicates the stream buffer input/output state
	openmode	Type that indicates the open mode of the file
	seekdir	Type that indicates the seek state of the stream buffer

Variable	fmtfl	Format flag
	wide	Field width
	prec	Precision (number of decimal point digits) at output
	fillch	Fill character
Function	void _ec2p_init_base()	Initializes the base class
	void _ec2p_copy_base(ios_base&ios_base_dt)	Copies ios_base_dt
	ios_base()	Constructor
	~ios_base()	Destructor
	fmtflags flags() const	References the format flag (fmtfl)
	fmtflags flags(fmtflags fmtflg)	Sets fmtflg&format flag (fmtfl) to the format flag (fmtfl)
	fmtflags setf(fmtflags fmtflg)	Sets fmtflg to format flag (fmtfl)
	fmtflags setf(fmtflags fmtflg, fmtflags mask)	Sets mask&fmtflg to the format flag (fmtfl)
	void unsetf(fmtflags mask)	Sets ~mask&format flag (fmtfl) to the format flag (fmtfl)
	char fill() const	References the fill character (fillch)
	char fill(char ch)	Sets ch as the fill character (fillch)
	int precision() const	References the precision (prec)
	streamsize precision(streamsize preci)	Sets preci as precision (prec)
	streamsize width() const	References the field width (wide)
	streamsize width(streamsize wd)	Sets wd as field width (wide)

ios_base::fmtflags

Defines the format control information relating to input/output processing. The definition for each bit mask of **fmtflags** is as follows:

const ios_base::fmtflags ios_base::boolalpha	= 0x0000;
const ios_base::fmtflags ios_base::skipws	= 0x0001;
const ios_base::fmtflags ios_base::unitbuf	= 0x0002;
const ios_base::fmtflags ios_base::uppercase	= 0x0004;
const ios_base::fmtflags ios_base::showbase	= 0x0008;
const ios_base::fmtflags ios_base::showpoint	= 0x0010;
const ios_base::fmtflags ios_base::showpos	= 0x0020;
const ios_base::fmtflags ios_base::left	= 0x0040;
const ios_base::fmtflags ios_base::right	= 0x0080;
const ios_base::fmtflags ios_base::internal	= 0x0100;
const ios_base::fmtflags ios_base::adjustfield	= 0x01c0;



const ios_base::fmtflags ios_base::dec	= 0x0200;
const ios_base::fmtflags ios_base::oct	= 0x0400;
const ios_base::fmtflags ios_base::hex	= 0x0800;
const ios_base::fmtflags ios_base::basefield	= 0x0e00;
const ios_base::fmtflags ios_base::scientific	= 0x1000;
const ios_base::fmtflags ios_base::fixed	= 0x2000;
const ios_base::fmtflags ios_base::floatfield	= 0x3000;
const ios_base::fmtflags ios_base::_fmtmask	= 0x3fff;

ios_base::iostate

Defines the input/output state of the stream buffer. The definition for each bit mask of **iostate** is as follows:

const ios_base::iostate ios_base::goodbit	= 0x0;
const ios_base::iostate ios_base::eofbit	= 0x1;
const ios_base::iostate ios_base::failbit	= 0x2;
const ios_base::iostate ios_base::badbit	= 0x4;
const ios_base::iostate ios_base::_statemask	= 0x7;

ios_base::openmode

Defines open mode of the file.

The definition for each bit mask of **openmode** is as follows:

const ios_base::openmode ios_base::in	= 0x01;	Opens the input file.
const ios_base::openmode ios_base::out	= 0x02;	Opens the output file.
const ios_base::openmode ios_base::ate	= 0x04;	Seeks for eof only once after the file has been opened.
const ios_base::openmode ios_base::app	= 0x08;	Seeks for eof each time the file is written to.
const ios_base::openmode ios_base::trunc	= 0x10;	Opens the file in overwrite mode.
const ios_base::openmode ios_base::binary	= 0x20;	Opens the file in binary mode.

ios_base::seekdir

Defines the seek state of the stream buffer.

Determines the position in a stream to continue the input/output of data. The definition for each bit mask of **seekdir** is as follows:

const ios_base::seekdir ios_base::beg	= 0x0;
const ios_base::seekdir ios_base::cur	= 0x1;
const ios_base::seekdir ios_base::end	= 0x2;



void ios_base::_ec2p_init_base()

The initial settings are as follows: fmtfl = skipws | dec; wide = 0; prec = 6; fillch = ` `;

void ios_base::_ec2p_copy_base(ios_base& ios_base_dt)

Copies ios_base_dt.

ios_base::ios_base()

Constructor of class **ios_base**. Calls **Init::Init()**.

ios_base::~ios_base()

Destructor of class ios_base.

ios_base::fmtflags ios_base::flags() const

References the format flag (fmtfl). Return value: Format flag (fmtfl)

ios_base::fmtflags ios_base::flags(fmtflags fmtflg)

Sets **fmtflg**&format flag (**fmtfl**) to the format flag (**fmtfl**). Return value: Format flag (**fmtfl**) before setting

ios_base::fmtflags ios_base::setf(fmtflags fmtflg)

Sets **fmtflg** to the format flag (**fmtfl**). Return value: Format flag (**fmtfl**) before setting

ios_base::fmtflags ios_base::setf((fmtflags fmtflg, fmtflags mask)

Sets the **mask&fmtflg** value to the format flag (**fmtfl**). Return value: Format flag (**fmtfl**) before setting.

void ios_base::unsetf(fmtflags mask)

Sets ~mask&format flag (fmtfl) to the format flag (fmtfl).

char ios_base::fill() const

References the fill character (fillch). Return value: Fill character (fillch)

char ios_base::fill(char ch)

Sets ch as the fill character (fillch).



Return value: Fill character (fillch) before setting

int ios_base::precision() const

References the precision (**prec**). Return value: Precision (**prec**)

streamsize ios_base::precision(streamsize preci)

Sets **preci** as the precision (**prec**). Return value: Precision (**prec**) before setting

streamsize ios_base::width() const

References the field width (**wide**). Return value: Field width (**wide**)

streamsize ios_base::width(streamsize wd)

Sets wd as the field width (wide). Return value: Field width (wide) before setting



(c) ios Class

Туре	Definition Name	Description
Variable	sb	Pointer to the streambuf object
	tiestr	Pointer to the ostream object
	state	State flag of streambuf
Function	ios()	Constructor
	ios(streambuf* sbptr)	
	void init(streambuf* sbptr)	Performs initial setting
	virtual ~ios()	Destructor
	operator void*() const	Tests whether an error has been generated (!state&(badbit failbit)
	bool operator!() const	Tests whether an error has been generated (state&(badbit failbit))
	iostate rdstate() const	References the state flag (state)
	void clear(iostate st = goodbit)	Clears the state flag (state) except for the specified state (st)
	void setstate(iostate st)	Specifies st as the state flag (state)
	bool good() const	Tests whether an error has been generated (state==goodbit)
	bool eof() const	Tests for the end of an input stream (state&eofbit)
	bool bad() const	Tests whether an error has been generated (state&badbit)
	bool fail() const	Tests whether the input text matches the requested pattern (state&(badbit failbit))
	ostream* tie() const	References the pointer to the ostream object (tiestr)
	ostream* tie(ostream* tstrptr)	Sets tstrptr as the pointer to the ostream object (tiestr)
	streambuf* rdbuf() const	References the pointer to the streambuf object (sb)
	streambuf* rdbuf(streambuf* sbptr)	Sets sbptr as the pointer to the streambuf object (sb)
	ios& copyfmt(const ios& rhs)	Copies the state flag (state) of rhs

ios::ios()

Constructor of class **ios**. Calls **init(0)** and sets the initial value to the member object.

ios::ios(streambuf* sbptr)

Constructor of class **ios**.

Calls init(sbptr) and sets the initial value to the member object.

void ios::init(streambuf* sbptr)

Sets sbptr to sb.



Sets state and tiestr to 0.

virtual ios::~ios()

Destructor of class ios.

ios::operator void*() const

Tests whether an error has been generated (!state&(badbit | failbit)). Return value: An error has been generated: false No error has been generated: true

bool ios::operator!() const

Tests whether an error has been generated (state&(badbit | failbit)). Return value: An error has been generated: true No error has been generated: false

iostate ios::rdstate() const

References the state flag (**state**). Return value: State flag (**state**)

void ios::clear(iostate st = goodbit)

Clears the state flag (state) except for the specified state (st). If the pointer to the streambuf object (sb) is 0, badbit is set to the state flag (state).

void ios::setstate(iostate st)

Sets st to the state flag (state).

bool ios::good() const

Tests whether an error has been generated (state==goodbit). Return value: An error has been generated: false No error has been generated: true

bool ios::eof() const

Tests for the end of the input stream (**state&eofbit**). Return value: End of the input stream has been reached: **true** End of the input stream has not been reached: **false**

bool ios::bad() const

Tests whether an error has been generated (**state&badbit**). Return value: An error has been generated: **true** No error has been generated: **false**

bool ios::fail() const

Tests whether the input text matches the requested pattern (state&(badbit | failbit)).



Return value: Does not match the requested pattern: true Matches the requested pattern: false

ostream* ios::tie() const

References the pointer (**tiestr**) to the **ostream** object. Return value: Pointer to the **ostream** object (**tiestr**)

ostream* ios::tie(ostream* tstrptr)

Sets **tstrptr** as the pointer (**tiestr**) to the **ostream** object. Return value: Pointer to the **ostream** object (**tiestr**) before setting

streambuf* ios::rdbuf() const

References the pointer to the **streambuf** object (**sb**). Return value: Pointer to the **streambuf** object (**sb**)

streambuf* ios::rdbuf(streambuf* sbptr)

Sets **sbptr** as the pointer to the **streambuf** object (**sb**). Return value: Pointer to the **streambuf** object (**sb**) before setting

ios& ios::copyfmt(const ios& rhs)

Copies the state flag (state) of rhs. Return value: *this



(d) ios Class Manipulators

Туре	Definition Name	Description
Function	ios_base& showbase(ios_base& str)	Specifies the radix display prefix mode
	ios_base& noshowbase(ios_base& str)	Clears the radix display prefix mode
	ios_base& showpoint(ios_base& str)	Specifies the decimal-point generation mode
	ios_base& noshowpoint(ios_base& str)	Clears the decimal-point generation mode
	ios_base& showpos(ios_base& str)	Specifies the + sign generation mode
	ios_base& noshowpos(ios_base& str)	Clears the + sign generation mode
	ios_base& skipws(ios_base& str)	Specifies the space skipping mode
	ios_base& noskipws (ios_base& str)	Clears the space skipping mode
	ios_base& uppercase(ios_base& str)	Specifies the uppercase letter conversion mode
	ios_base& nouppercase(ios_base& str)	Clears the uppercase letter conversion mode
	ios_base& internal(ios_base& str)	Specifies the internal fill mode
	ios_base& left(ios_base& str)	Specifies the left side fill mode
	ios_base& right(ios_base& str)	Specifies the right side fill mode
	ios_base& dec(ios_base& str)	Specifies the decimal mode
	ios_base& hex(ios_base& str)	Specifies the hexadecimal mode
	ios_base& oct(ios_base& str)	Specifies the octal mode
	ios_base& fixed(ios_base& str)	Specifies the fixed-point mode
	ios_base& scientific(ios_base& str)	Specifies the scientific description mode

ios_base& showbase(ios_base& str)

Specifies an output mode of prefixing a radix at the beginning of data. For a hexadecimal, 0x is prefixed. For a decimal, nothing is prefixed. For an octal, 0 is prefixed. Return value: **str**

ios_base& noshowbase(ios_base& str)

Clears the output mode of prefixing a radix at the beginning of data. Return value: $\ensuremath{\text{str}}$

ios_base& showpoint(ios_base& str)

Specifies the output mode of showing the decimal point. If no precision is specified, six decimal-point (fraction) digits are displayed. Return value: **str**

ios_base& noshowpoint(ios_base& str)

Clears the output mode of showing the decimal point. Return value: **str**



ios_base& showpos(ios_base& str)

Specifies the output mode of generating the + sign (adds a + sign to a positive number). Return value: **str**

ios_base& noshowpos(ios_base& str)

Clears the output mode of generating the + sign. Return value: **str**

ios_base& skipws(ios_base& str)

Specifies the input mode of skipping spaces (skips consecutive spaces). Return value: **str**

ios_base& noskipws(ios_base& str)

Clears the input mode of skipping spaces. Return value: **str**

ios_base& uppercase(ios_base& str)

Specifies the output mode of converting letters to uppercases.

In hexadecimal, the radix will be uppercase letters 0X, and the numeric value letters will be uppercase letters. The exponential representation of a floating-point value will also use uppercase letter E.

Return value: str

ios_base& nouppercase(ios_base& str)

Clears the output mode of converting letters to uppercases. Return value: **str**

ios_base& internal(ios_base& str)

When data is output in the field width (**wide**) range, it is output in the order of Sign and radix Fill character (**fill**) Numeric value Return value: **str**

ios_base& left(ios_base& str)

When data is output in the field width (**wide**) range, it is aligned to the left. Return value: **str**

ios_base& right(ios_base& str)

When data is output in the field width (wide) range, it is aligned to the right. Return value: $\ensuremath{\text{str}}$

ios_base& dec(ios_base& str)

Specifies the conversion radix to the decimal mode. Return value: **str**



ios_base& hex(ios_base& str)

Specifies the conversion radix to the hexadecimal mode. Return value: $\ensuremath{\textit{str}}$

ios_base& oct(ios_base& str)

Specifies the conversion radix to the octal mode. Return value: **str**

ios_base& fixed(ios_base& str)

Specifies the fixed-point output mode. Return value: **str**

ios_base& scientific(ios_base& str)

Specifies the scientific description output mode (exponential description). Return value: $\ensuremath{\text{str}}$



(e) streambuf Class

Туре	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 .



Туре	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 ${\bf n}$ characters and sets them in the memory area specified by ${\bf s}$
	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 ${\bf n}$ characters at the position pointed to by the amount specified by ${\bf s}$
	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 ${\bf n}$
	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 ${\bf n}$
-	void setp(char* pbeg, char* pend)	Assigns each pointer of the output stream



Туре	Definition Name	Description
Function	virtual streambuf* setbuf(char* s, streamsize n)* ¹	For each derived class, a defined opera- tion is executed
	<pre>virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in ios_base::out))*1</pre>	Changes the stream position
	<pre>virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in ios_base::out))*1</pre>	Changes the stream position
	virtual int sync()* ¹	Flushes the output stream
	virtual int showmanyc()* ¹	Calculates the number of valid charac- ters 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 ${\bf n}$ characters in the position specified by ${\bf s}$
	virtual int_type overflow(int type c = eof)* ¹	Inserts character ${f c}$ in the output stream

Note This class does not define the processing.

streambuf::streambuf()

Constructor.

The initial settings are as follows: _B_cnt_ptr = B_beg_ptr = B_next_ptr = B_end_ptr = C_flg_ptr = _B_len_ptr = 0 B_beg_pptr = &B_beg_ptr B_next_pptr = &B_next_ptr

virtual streambuf::~streambuf()

Destructor.

streambuf* streambuf::pubsetbuf(char* s, streamsize n)

Allocates the buffer for stream input/output. This function calls setbuf (s,n). Return value: *this

pos_type streambuf::pubseekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = (ios_base::openmode)(ios_base::in | ios_base::out))

Moves the read or write position for the input/output stream by using the method specified by way. This function calls seekoff(off,way,which).

Return value: The stream position newly specified



pos_type streambuf::pubseekpos(pos_type sp, ios_base::openmode which = (ios_base::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

int streambuf::pubsync()

Flushes the output stream. This function calls **sync()**. Return value: 0

streamsize streambuf::in_avail()

Calculates the offset from the end of the input stream to the current position. Return value:

If the position where data is read is valid: The offset from the end of the stream to the current position If the position where data is read is invalid: 0 (**showmanyc()** is called)

int_type streambuf::snextc()

Reads one character. If the character read is not **eof**, the next character is read. Return value: If the character read is not **eof**: The character read If the character read is **eof**: **eof**

int_type streambuf::sbumpc()

Reads one character and moves forward the pointer to the next. Return value: If the position where data is read is valid: The character read If the position where data is read is invalid: **eof**

int_type streambuf::sgetc()

Reads one character.

Return value: If the position where data is read is valid: The character read If the position where data is read is invalid: **eof**

int streambuf::sgetn(char* s, streamsize n)

Sets **n** characters in the memory area specified by **s**. If an **eof** is found in the string read, setting is stopped. Return value: The specified number of characters

int_type streambuf::sputbackc(char c)

If the data read position is correct and the put back data of the position is the same as \mathbf{c} , the read position is put back. Return value: If the read position was put back: The value of \mathbf{c}

If the read position was not put back: eof

int streambuf::sungetc()

If the data read position is correct, the read position is put back. Return value: If the read position was put back: The value that was put back If the read position was not put back: **eof**

RENESAS

int streambuf::sputc(char c)

Inserts character c.

Return value: If the write position is correct: The value of **c** If the write position is incorrect: **eof**

int_type streambuf::sputn(const char* s, streamsize n)

Inserts **n** characters at the position specified by **s**. If the buffer is smaller than **n**, the number of characters for the buffer is inserted. Return value: The number of characters inserted

char* streambuf::eback() const

Calculates the start pointer of the input stream. Return value: Start pointer

char* streambuf::gptr() const

Calculates the next pointer of the input stream. Return value: Next pointer

char* streambuf::egptr() const

Calculates the end pointer of the input stream. Return value: End pointer

void streambuf::gbump(int n)

Moves forward the next pointer of the input stream by the amount specified by n.

void streambuf::setg(char* gbeg, char* gnext, char* gend)

Sets each pointer of the input stream as follows:

*B_beg_pptr = gbeg; *B_next_pptr = gnext; B_end_ptr = gend; *_B_cnt_ptr = gend-gnext;

*_B_len_ptr = gend-gbeg;

char* streambuf::pbase() const

Calculates the start pointer of the output stream. Return value: Start pointer

char* streambuf::pptr() const

Calculates the next pointer of the output stream. Return value: Next pointer

char* streambuf::epptr() const

Calculates the end pointer of the output stream. Return value: End pointer



void streambuf::pbump(int n)

Moves forward the next pointer of the output stream by the amount specified by n.

void streambuf::setp(char* pbeg, char* pend)

The settings for each pointer of the output stream are as follows:

*B_beg_pptr = pbeg; *B_next_pptr = pbeg; B_end_ptr = pend; *_B_cnt_ptr = pend-pbeg; *_B_len_ptr = pend-pbeg;

virtual streambuf* streambuf::setbuf(char* s, streamsize n)

For each derived class from **streambuf**, a defined operation is executed. Return value: ***this** (This class does not define the processing.)

virtual pos_type streambuf::seekoff(off_type off, ios_base::seekdir way, ios_base::openmode =
(ios_base::openmode)(ios_base::in | ios_base::out))

Changes the stream position.

Return value: -1 (This class does not define the processing.)

virtual pos_type streambuf::seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode)(ios_base::in |
ios_base::out))

Changes the stream position.

Return value: -1 (This class does not define the processing.)

virtual int streambuf::sync()

Flushes the output stream. Return value: 0 (This class does not define the processing.)

virtual int streambuf::showmanyc()

Calculates the number of valid characters in the input stream. Return value: 0 (This class does not define the processing.)

virtual streamsize streambuf::xsgetn(char* s, streamsize n)

Sets **n** characters in the memory area specified by **s**. If the buffer is smaller than **n**, the number of characters for the buffer is inserted. Return value: The number of characters input

virtual int_type streambuf::underflow()

Reads one character without moving the stream position. Return value: **eof** (This class does not define the processing.)

virtual int_type streambuf::uflow()

Reads one character of the next pointer.



Return value: **eof** (This class does not define the processing.)

virtual int_type streambuf::pbackfail(int_type c = eof)

Puts back the character specified by **c**.

Return value: **eof** (This class does not define the processing.)

virtual streamsize streambuf::xsputn(const char* s, streamsize n)

Inserts **n** characters specified by **s** in to the stream position. If the buffer is smaller than **n**, the number of characters for the buffer is inserted. Return value: The number of characters inserted

virtual int_type streambuf::overflow(int_type c = eof)

Inserts character **c** in the output stream. Return value: **eof** (This class does not define the processing.)

(f) istream::sentry Class

Туре	Definition Name	Description
Variable	ok_	Whether the current state is input-enabled
Function	sentry(istream& is, bool noskipws = false)	Constructor
	~sentry()	Destructor
	operator bool()	References ok_

istream::sentry::sentry(istream& is, bool noskipws = _false)

Constructor of internal class sentry.

If **good()** is non-zero, enables input with or without a format. If **tie()** is non-zero, flushes the related output stream.

istream::sentry::~sentry()

Destructor of internal class sentry.

istream::sentry::operator bool()

References **ok_**. Return value: **ok_**



(g) istream Class

Туре	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
	istream& operator>>(short& n)	n 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



Туре	Definition Name	Description
Function	istream& get(char& c)	Extracts characters and stores them in $\ensuremath{\textbf{c}}$
	istream& get(signed char& c)	
	istream& get(unsigned char& c)	
	istream& get(char* s, streamsize n)	Extracts strings with size n-1 and stores
	istream& get(signed char* s, streamsize n)	them in the memory area specified by s
	istream& get(unsigned char* s, streamsize n)	
	istream& get(char* s, streamsize n, char delim)	Extracts strings with size n-1 and stores
	istream& get(signed char* s, streamsize n, char delim)	If delim is found in the string, input is stopped.
	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
	istream& getline(signed char* s, streamsize n)	them in the memory area specified by s .
	istream& getline(unsigned char* s, streamsize n)	
	istream& getline(char* s, streamsize n, char delim)	Extracts strings with size n-1 and stores
	istream& getline(signed char* s, streamsize n, char delim)	If delim is found in the string, input is stopped.
	istream& getline(unsigned char* s, streamsize n, char delim)	



Туре	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
	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
	streamsize readsome(signed char* s, streamsize n)	them in the memory area specified by s
	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::pub- sync() .
	pos_type tellg()	Finds the input stream position. This function calls streambuf::pub- seekoff(0,cur,in) .
	istream& seekg(pos_type pos)	Moves the current stream pointer by the amount specified by pos . This function calls streambuf::pub- seekpos(pos) .
	istream& seekg(off_type off, ios_base::seekdir dir)	Moves the position to read the input stream by using the method specified by dir . This function calls streambuf::pub- seekoff(off,dir) .

int istream::_ec2p_getistr(char* str, unsigned int dig, int mode)

Converts **str** to the radix specified by **dig**. Return value: The converted radix

istream::istream(streambuf* sb)

Constructor of class **istream**. Calls **ios::init(sb)**. Specifies **chcount=**0.

virtual istream::~istream()

Destructor of class istream.



istream& istream::operator>>(bool& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(short& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(unsigned short& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(int& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(unsigned int& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(long& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(unsigned long& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(long long& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(unsigned long long& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(float& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(double& n)

Stores the extracted characters in n.



Return value: *this

istream& istream::operator>>(long double& n)

Stores the extracted characters in **n**. Return value: ***this**

istream& istream::operator>>(void*& p)

Converts the extracted characters to a **void*** type and stores them in the memory specified by **p**. Return value: ***this**

istream& istream::operator>>(streambuf* sb)

Extracts characters and stores them in the memory area specified by **sb**. If there are no extracted characters, **setstate(failbit)** is called. Return value: ***this**

streamsize istream::gcount() const

References **chcount** (number of extracted characters). Return value: **chcount**

int_type istream::get()

Extracts characters.

Return value: If characters are extracted: Extracted characters. If no characters are extracted: Calls **setstate(failbit)** and becomes **streambuf::eof**.

istream& istream::get(char& c)

Extracts characters and stores them in **c**. If the extracted character is **streambuf::eof**, **failbit** is set. Return value: ***this**

istream& istream::get(signed char& c)

Extracts characters and stores them in **c**. If the extracted character is **streambuf::eof**, **failbit** is set. Return value: ***this**

istream& istream::get(unsigned char& c)

Extracts characters and stores them in c. If the extracted character is streambuf::eof, failbit is set. Return value: *this

istream& istream::get(char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **ok_==false** or no character has been extracted, **failbit** is set.

Return value: *this

istream& istream::get(signed char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **ok_==false** or no character has been extracted, **failbit** is set.



Return value: *this

istream& istream::get(unsigned char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **ok_==false** or no character has been extracted, **failbit** is set.

Return value: *this

istream& istream::get(char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **delim** is found in the string, input is stopped. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::get(signed char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **delim** is found in the string, input is stopped. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::get(unsigned char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **delim** is found in the string, input is stopped. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::get(streambuf& sb)

Extracts a string and stores it in the memory area specified by **sb**. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::get(streambuf& sb, char delim)

Extracts a string and stores it in the memory area specified by **sb**. If **delim** is found in the string, input is stopped. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::getline(char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::getline(signed char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**



istream& istream::getline(unsigned char* s, streamsize n)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::getline(char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If character **delim** is found, input is stopped. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::getline(signed char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If character **delim** is found, input is stopped. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::getline(unsigned char* s, streamsize n, char delim)

Extracts a string with size **n-1** and stores it in the memory area specified by **s**. If character **delim** is found, input is stopped. If **ok_==false** or no character has been extracted, **failbit** is set. Return value: ***this**

istream& istream::ignore(streamsize n = 1, int type delim = streambuf::eof)

Skips reading the number of characters specified by **n**. If character **delim** is found, skipping is stopped. Return value: ***this**

int_type istream::peek()

Seeks input characters that will be available next. Return value: If ok_==false: streambuf::eof If ok_!=false: rdbuf()->sgetc()

istream& istream::read(char* s, streamsize n)

If **ok_!=false**, extracts a string with size **n** and stores it in the memory area specified by **s**. If the number of extracted characters does not match with the number of **n**, **eofbit** is set. Return value: ***this**

istream& istream::read(signed char* s, streamsize n)

If **ok_!=false**, extracts a string with size **n** and stores it in the memory area specified by **s**. If the number of extracted characters does not match with the number of **n**, **eofbit** is set. Return value: ***this**

istream& istream::read(unsigned char* s, streamsize n)

If **ok_!=false**, extracts a string with size **n** and stores it in the memory area specified by **s**. If the number of extracted characters does not match with the number of **n**, **eofbit** is set.



Return value: *this

streamsize istream::readsome(char* s, streamsize n)

Extracts a string with size **n** and stores it in the memory area specified by **s**. If the number of characters exceeds the stream size, only the number of characters equal to the stream size is stored. Return value: The number of extracted characters

streamsize istream::readsome(signed char* s, streamsize n)

Extracts a string with size **n** and stores it in the memory area specified by **s**. If the number of characters exceeds the stream size, only the number of characters equal to the stream size is stored. Return value: The number of extracted characters

streamsize istream::readsome(unsigned char* s, streamsize n)

Extracts a string with size **n** and stores it in the memory area specified by **s**. If the number of characters exceeds the stream size, only the number of characters equal to the stream size is stored. Return value: The number of extracted characters

istream& istream::putback(char c)

Puts back character **c** to the input stream. If the characters put back are **streambuf::eof**, **badbit** is set. Return value: ***this**

istream& istream::unget()

Puts back the pointer of the input stream by one. If the extracted characters are **streambuf::eof**, **badbit** is set.

Return value: *this

int istream::sync()

Checks for an input stream. This function calls **streambuf::pubsync()**. Return value: If there is no input stream: **streambuf::eof** If there is an input stream: 0

pos_type istream::tellg()

Checks for the position of the input stream. This function calls **streambuf::pubseekoff(0,cur,in)**. Return value: Offset from the beginning of the stream If an error occurs during the input processing, -1 is returned.

istream& istream::seekg(pos_type pos)

Moves the current stream pointer by the amount specified by **pos**. This function calls **streambuf::pubseekpos(pos)**. Return value: ***this**

istream& istream::seekg(off_type off, ios_base::seekdir dir)

Moves the position to read the input stream using the method specified by dir.



This function calls **streambuf::pubseekoff(off,dir)**. If an error occurs during the input processing, this processing is not performed.

Return value: *this

(h) istream Class Manipulator

Туре	Definition Name	Description
Function	istream& ws(istream& is)	Skips reading the spaces

istream& ws(istream& is)

Skips reading white spaces. Return value: **is**

(i) istream Non-Member Function

Туре	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	
	istream& operator>>(istream& in, signed char& c)	it in c	
	istream& operator>>(istream& in, unsigned char& c)		

istream& operator>>(istream& in, char* s)

Extracts a string and stores it in the memory area specified by ${\bf s}.$ Processing is stopped if

the number of characters stored is equal to field width – 1

streambuf::eof is found in the input stream

the next available character c satisfies isspace(c)==1

If no characters are stored, failbit is set.

Return value: in

istream& operator>>(istream& in, signed char* s)

Extracts a string and stores it in the memory area specified by **s**. Processing is stopped if the number of characters stored is equal to field width – 1

streambuf::eof is found in the input stream
the next available character c satisfies isspace(c)==1
If no characters are stored, failbit is set.
Return value: in

istream& operator>>(istream& in, unsigned char* s)

Extracts a string and stores it in the memory area specified by **s**. Processing is stopped if the number of characters stored is equal to field width – 1 **streambuf::eof** is found in the input stream the next available character **c** satisfies **isspace(c)==1** If no characters are stored, **failbit** is set. Return value: **in**



istream& operator>>(istream& in, char& c)

Extracts a character and stores it in **c**. If no character is stored, **failbit** is set. Return value: **in**

istream& operator>>(istream& in, signed char& c)

Extracts a character and stores it in **c**. If no character is stored, **failbit** is set. Return value: **in**

istream& operator>>(istream& in, unsigned char& c)

Extracts a character and stores it in ${\bf c}.$ If no character is stored, ${\bf failbit}$ is set. Return value: ${\bf in}$

(j) ostream::sentry Class

Туре	Definition Name	Description
Variable	ok_	Whether or not the current state allows output
	ec2p_os	Pointer to the ostream object
Function	sentry(ostream& os)	Constructor
	~sentry()	Destructor
	operator bool()	References ok_

ostream::sentry::sentry(ostream& os)

Constructor of the internal class **sentry**. If **good()** is non-zero and **tie()** is non-zero, **flush()** is called.

Specifies os to _ _ec2p_os.

ostream::sentry::~sentry()

Destructor of internal class sentry.

If (__ec2p_os->flags() & ios_base::unitbuf) is true, flush() is called.

ostream::sentry::operator bool()

References **ok_**. Return value: **ok_**



(k) ostream Class

Туре	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 ${f c}$ into the output stream.
	ostream& write(const char* s, streamsize n)	Inserts n characters from s into the output stream.
	ostream& write(const signed char* s, streamsize n)	
	ostream& write(const unsigned char* s, streamsize n)	
	ostream& flush()	Flushes the output stream. This function calls streambuf::pubsync() .
	pos_type tellp()	Calculates the current write position. This function calls streambuf::pub- seekoff(0,cur,out) .
	ostream& seekp(pos_type pos)	Calculates the offset from the beginning of the stream to the current position. Moves the current stream pointer by the amount specified by pos . This function calls streambuf::pubseek- pos(pos) .
	ostream& seekp(off_type off, seekdir dir)	Moves the stream write position by the amount specified by off , from dir . This function calls streambuf::pub-seekoff(off,dir) .



ostream::ostream(streambuf* sbptr)

Constructor. Calls **ios(sbptr)**.

virtual ostream::~ostream()

Destructor.

ostream& ostream::operator<<(bool n)

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(short n)</pre>

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(unsigned short n)

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(int n)

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(unsigned int n)</pre>

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(long n)

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(unsigned long n)

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(long long n)



If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(unsigned long long n)</pre>

If **sentry::ok_==true**, **n** is inserted into the output stream. If **sentry::ok_==false**, **failbit** is set. Return value: ***this**

ostream& ostream::operator<<(float n)

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(double n)

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(long double n)

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(void* n)

If sentry::ok_==true, n is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::operator<<(streambuf* sbptr)

If sentry::ok_==true, the output string of sbptr is inserted into the output stream. If sentry::ok_==false, failbit is set. Return value: *this

ostream& ostream::put(char c)

If (sentry::ok_==true) and (rdbuf()->sputc(c)!=streambuf::eof), c is inserted into the output stream. Otherwise badbit is set. Return value: *this

ostream& ostream::write(const char* s, streamsize n)

If (sentry::ok_==true) and (rdbuf()->sputn(s, n)==n), n characters specified by s are inserted into the output stream. Otherwise **badbit** is set. Return value: *this

ostream& ostream::write(const signed char* s, streamsize n)

If (sentry::ok_==true) and (rdbuf()->sputn(s, n)==n), n characters specified by s are inserted into the output stream.



Otherwise **badbit** is set. Return value: ***this**

ostream& ostream::write(const unsigned char* s, streamsize n)

If (sentry::ok_==true) and (rdbuf()->sputn(s, n)==n), n characters specified by s are inserted into the output stream. Otherwise **badbit** is set. Return value: *this

ostream& ostream::flush()

Flushes the output stream. This function calls **streambuf::pubsync()**. Return value: ***this**

pos_type ostream::tellp()

Calculates the current write position. This function calls **streambuf::pubseekoff(0,cur,out)**. Return value: The current stream position If an error occurs during processing, -1 is returned.

ostream& ostream::seekp(pos_type pos)

If no error occurs, the offset from the beginning of the stream to the current position is calculated. Moves the current stream pointer by the amount specified by **pos**. This function calls **streambuf::pubseekpos(pos)**. Return value: ***this**

ostream& ostream::seekp(off_type off, seekdir dir)

If no error occurs, the stream write position is moved by the amount specified by **off**, from **dir**. This function calls **streambuf::pubseekoff(off,dir)**. Return value: ***this**

(I) ostream Class Manipulator

Туре	Definition Name	Description
Function	ostream& endl(ostream& os)	Inserts a new line and flushes the output stream
	ostream& ends(ostream& os)	Inserts a NULL code
	ostream& flush(ostream& os)	Flushes the output stream

ostream& endl(ostream& os)

Inserts a new line code and flushes the output stream. This function calls **flush()**. Return value: **os**

ostream& ends(ostream& os)

Inserts a **NULL** code into the output line. Return value: **os**



ostream& flush(ostream& os)

Flushes the output stream. This function calls **streambuf::sync()**. Return value: **os**

(m) ostream Non-Member Function

Туре	Definition Name	Description
Function	ostream& operator<<(ostream& os, char s)	Inserts s into the output stream
	ostream& operator<<(ostream& os, signed char s)	
	ostream& operator<<(ostream& os, unsigned char s)	
	ostream& operator<<(ostream& os, const char* s)	
	ostream& operator<<(ostream& os, const signed char* s)	
	ostream& operator<<(ostream& os, const unsigned char* s)	

ostream& operator<<(ostream& os, char s)

If (sentry::ok_==true) and an error does not occur, s is inserted into the output stream. Otherwise failbit is set. Return value: os

ostream& operator<<(ostream& os, signed char s)

If (sentry::ok_==true) and an error does not occur, s is inserted into the output stream. Otherwise failbit is set. Return value: os

ostream& operator<<(ostream& os, unsigned char s)

If (sentry::ok_==true) and an error does not occur, s is inserted into the output stream. Otherwise failbit is set. Return value: os

ostream& operator<<(ostream& os, const char* s)

If (sentry::ok_==true) and an error does not occur, s is inserted into the output stream. Otherwise failbit is set. Return value: os

ostream& operator<<(ostream& os, const signed char* s)

If (sentry::ok_==true) and an error does not occur, s is inserted into the output stream. Otherwise failbit is set. Return value: os

ostream& operator<<(ostream& os, const unsigned char* s)

If (sentry::ok_==true) and an error does not occur, s is inserted into the output stream. Otherwise failbit is set. Return value: os


(n) smanip Class Manipulator

Туре	Definition Name	Description
Function	smanip resetiosflags(ios_base::fmtflags mask)	Clears the flag specified by the mask value
	smanip setiosflags(ios_base::fmtflags mask)	Specifies the format flag (fmtfl)
	smanip setbase(int base)	Specifies the radix used at output
	smanip setfill(char c)	Specifies the fill character (fillch)
	smanip setprecision(int n)	Specifies the precision (prec)
	smanip setw(int n)	Specifies the field width (wide)

smanip resetiosflags(ios_base::fmtflags mask)

Clears the flag specified by the **mask** value. Return value: Target object of input/output

smanip setiosflags(ios_base::fmtflags mask)

Specifies the format flag (**fmtfl**). Return value: Target object of input/output

smanip setbase(int base)

Specifies the radix used at output. Return value: Target object of input/output

smanip setfill(char c)

Specifies the fill character (**fillch**). Return value: Target object of input/output

smanip setprecision(int n)

Specifies the precision (**prec**). Return value: Target object of input/output

smanip setw(int n)

Specifies the field width (wide).

Return value: Target object of input/output

 (o) Example of Using EC++ Input/Output Libraries
 The input/output stream can be used if a pointer to an object of the mystrbuf class is used instead of streambuf at the initialization of the istream and ostream objects.
 The following shows the inheritance relationship of the above classes. An arrow (->) indicates that a derived class references a base class.





Туре	Definition Name	Description
Variable	_file_Ptr	File pointer.
Function	mystrbuf()	Constructor.
	mystrbuf(void* ptr)	initializes the streambur buffer.
	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.
	<pre>virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in ios_base::out))</pre>	Changes the position of the stream pointer.
	<pre>virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in ios_base::out))</pre>	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> oid main(void)</stdio.h></new></string></mystrbuf></ostream></istream>	
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;	

6.5.2 Memory Management Library

The header file for the memory management library is as follows:

- <new>

Defines the memory allocation/deallocation function.

By setting an exception handling function address to the **_ec2p_new_handler** variable, exception handling can be executed if memory allocation fails. The **_ec2p_new_handler** is a **static** variable and the initial value is **NULL**. If this handler is used, reentrance will be lost.

Operations required for the exception handling function:

- Creates an allocatable area and returns the area.

- Operations are not prescribed for cases where an area cannot be created.

Туре	Definition Name	Description
Туре	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



void* operator new(size_t size)

Allocates a memory area with the size specified by **size**. If memory allocation fails and when **new_handler** is set, **new_handler** is called. Return value: If memory allocation succeeds: Pointer to **void** type If memory allocation fails: **NULL**

void* operator new[](size_t size)

Allocates an array area with the size specified by **size**. If memory allocation fails and when **new_handler** is set, **new_handler** is called. Return value: If memory allocation succeeds: Pointer to **void** type If memory allocation fails: **NULL**

void* operator new(size_t size, void* ptr)

Allocates the area specified by **ptr** as the storage area. Return value: **ptr**

void* operator new[](size_t size, void* ptr)

Allocates the area specified by **ptr** as the array area. Return value: **ptr**

void operator delete(void* ptr)

Deallocates the storage area specified by **ptr**. If **ptr** is **NULL**, no operation will be performed.

void operator delete[](void* ptr)

Deallocates the array area specified by **ptr**. If **ptr** is **NULL**, no operation will be performed.

new_handler set_new_handler(new_handler new_P)

Sets new_P to _ec2p_new_handler. Return value: _ec2p_new_handler



6.5.3 Complex Number Calculation Class Library

The header file for the complex number calculation class library is as follows:

- <complex>

Defines the **float_complex** and **double_complex** classes.

These classes have no derivation.

(a) float_complex Class

Туре	Definition Name	Description
Туре	value_type	float type
Variable	_re	Defines the real part of float precision
	_im	Defines the imaginary part of float precision
Function	float_complex(float re = 0.0f, float im = 0.0f)	Constructor
	float_complex(const double_complex& rhs)	
	float real() const	Acquires the real part (_ re)
	float imag() const	Acquires the imaginary part (_im)
	float_complex& operator=(float rhs)	Copies rhs to the real part. 0.0f is assigned to the imaginary part.
	float_complex& operator+=(float rhs)	Adds rhs to the real part and stores the sum in *this .
	float_complex& operator-=(float rhs)	Subtracts rhs from the real part and stores the difference in *this .
	float_complex& operator*=(float rhs)	Multiplies *this by rhs and stores the product in *this .
	float_complex& operator/=(float rhs)	Divides *this by rhs and stores the quotient in *this .
	float_complex& operator=(const float_complex& rhs)	Copies rhs .
-	float_complex& operator+=(const float_complex& rhs)	Adds rhs to *this and stores the sum in *this .
	float_complex& operator-=(const float_complex& rhs)	Subtracts rhs from *this and stores the difference in *this .
	float_complex& operator*=(const float_complex& rhs)	Multiplies *this by rhs and stores the product in *this .
	float_complex& operator/=(const float_complex& rhs)	Divides *this by rhs and stores the quotient in *this .

float_complex::float_complex(float re = 0.0f, float im = 0.0f)

Constructor of class float_complex.

The initial settings are as follows:

_re = re; _im = im;

float_ complex::float_complex(const double_complex& rhs)

Constructor of class **float_complex**. The initial settings are as follows:



_re = (float)rhs.real();

_im = (float)rhs.imag();

float float_complex::real() const

Acquires the real part. Return value: **this->_re**

float float_complex::imag() const

Acquires the imaginary part. Return value: **this->_im**

float_complex& float_complex::operator=(float rhs)

Copies **rhs** to the real part (**_re**). 0.0f is assigned to the imaginary part (**_im**). Return value: ***this**

float_complex& float_complex::operator+=(float rhs)

Adds **rhs** to the real part (**_re**) and stores the result in the real part (**_re**). The value of the imaginary part (**_im**) does not change. Return value: ***this**

float_complex& float_complex::operator-=(float rhs)

Subtracts **rhs** from the real part (_**re**) and stores the result in the real part (_**re**). The value of the imaginary part (_**im**) does not change. Return value: ***this**

float_complex& float_complex::operator*=(float rhs)

Multiplies *this by rhs and stores the result in *this. (_re=_re*rhs, _im=_im*rhs) Return value: *this

float_complex& float_complex::operator/=(float rhs)

Divides ***this** by **rhs** and stores the result in ***this**. (_re=_re/rhs, _im=_im/rhs) Return value: ***this**

float_complex& float_complex::operator=(const float_complex& rhs)

Copies **rhs** to ***this**. Return value: ***this**

float_complex& float_complex::operator+=(const float_complex& rhs)

Adds **rhs** to ***this** and stores the result in ***this** Return value: ***this**



float_complex& float_complex::operator-=(const float_complex& rhs)

Subtracts **rhs** from ***this** and stores the result in ***this**. Return value: ***this**

float_complex& float_complex::operator*=(const float_complex& rhs)

Multiplies ***this** by **rhs** and stores the result in ***this**. Return value: ***this**

float_complex& float_complex::operator/=(const float_complex& rhs)

Divides ***this** by **rhs** and stores the result in ***this**. Return value: ***this**

(b) float_complex Non-Member Function

Туре	Definition Name	Description
Function	float_complex operator+(const float_complex& lhs)	Performs unary + operation of Ihs
	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



Туре	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 Ihs by rhs
	float_complex operator/(const float_complex& lhs, const float& rhs)	
	float_complex operator/(const float& lhs, const float_complex& rhs)	Divides Ihs by rhs and stores the quotient in Ihs
	bool operator==(const float_complex& lhs, const float_complex& rhs)	Compares the real parts of Ihs and rhs , and the imaginary parts of Ihs and rhs
	bool operator==(const float_complex& lhs, const float& rhs)	
	bool operator== (const float& lhs, const float_complex& rhs)	



Туре	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



Туре	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

float_complex operator+(const float_complex& lhs)

Performs unary + operation of **Ihs**. Return value: **Ihs**

float_complex operator+(const float_complex& lhs, const float_complex& rhs)

Returns the result of adding **lhs** to **rhs**. Return value: **float_complex(lhs)**+=**rhs**

float_complex operator+(const float_complex& lhs, const float& rhs)

Returns the result of adding **lhs** to **rhs**. Return value: **float_complex(lhs)**+=**rhs**

float_complex operator+(const float& lhs, const float_complex& rhs)

Returns the result of adding **lhs** to **rhs**. Return value: **float_complex(lhs)**+**=rhs**

float_complex operator-(const float_complex& lhs)

Performs unary - operation of **lhs**. Return value: **float_complex(-lhs.real(), -lhs.imag())**

float_complex operator-(const float_complex& lhs, const float_complex& rhs)

Returns the result of subtracting **rhs** from **lhs**. Return value: **float_complex(lhs)-=rhs**



float complex operator-(const float complex& lhs, const float& rhs)

Returns the result of subtracting **rhs** from **lhs**. Return value: **float_complex(lhs)-=rhs**

float_complex operator-(const float& lhs, const float_complex& rhs)

Returns the result of subtracting **rhs** from **lhs**. Return value: **float_complex(lhs)-=rhs**

float_complex operator*(const float_complex& lhs, const float_complex& rhs)

Returns the result of multiplying **lhs** by **rhs**. Return value: **float_complex(lhs)*=rhs**

float_complex operator*(const float_complex& lhs, const float& rhs)

Returns the result of multiplying **lhs** by **rhs**. Return value: **float_complex(lhs)*=rhs**

float_complex operator*(const float& lhs, const float_complex& rhs)

Returns the result of multiplying **lhs** by **rhs**. Return value: **float_complex(lhs)*=rhs**

float_complex operator/(const float_complex& lhs, const float_complex& rhs)

Returns the result of dividing **lhs** by **rhs**. Return value: **float_complex(lhs)**/=**rhs**

float_complex operator/(const float_complex& lhs, const float& rhs)

Returns the result of dividing **lhs** by **rhs**. Return value: **float_complex(lhs)**/=**rhs**

float_complex operator/(const float& lhs, const float_complex& rhs)

Returns the result of dividing **lhs** by **rhs**. Return value: **float_complex(lhs)**/=**rhs**

bool operator==(const float_complex& lhs, const float_complex& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **float** type parameter, the imaginary part is assumed to be 0.0f. Return value: **Ihs.real()==rhs.real()** && **Ihs.imag()==rhs.imag()**

bool operator==(const float_complex& lhs, const float& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **float** type parameter, the imaginary part is assumed to be 0.0f. Return value: **Ihs.real()==rhs.real()** && **Ihs.imag()==rhs.imag()**



bool operator==(const float& lhs, const float_complex& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **float** type parameter, the imaginary part is assumed to be 0.0f. Return value: **Ihs.real()==rhs.real()** && **Ihs.imag()==rhs.imag()**

bool operator!=(const float_complex& lhs, const float_complex& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **float** type parameter, the imaginary part is assumed to be 0.0f. Return value: **Ihs.real()**!=**rhs.real()** || **Ihs.imag()**!=**rhs.imag()**

bool operator!=(const float_complex& lhs, const float& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **float** type parameter, the imaginary part is assumed to be 0.0f. Return value: **Ihs.real()**!=**rhs.real()** || **Ihs.imag()**!=**rhs.imag()**

bool operator!=(const float& lhs, const float_complex& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **float** type parameter, the imaginary part is assumed to be 0.0f. Return value: **Ihs.real()**!=**rhs.real()** || **Ihs.imag()**!=**rhs.imag()**

istream& operator>>(istream& is, float_complex& x)

Inputs **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part). The input value is converted to **float_complex**. If **x** is input in a format other than the **u**, (**u**), or (**u**,**v**) format, **is.setstate(ios_base::failbit)** is called. Return value: **is**

ostream& operator<<(ostream& os, const float_complex& x)</pre>

Outputs **x** to **os**. The output format is **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part). Return value: **os**

float real(const float_complex& x)

Acquires the real part. Return value: **x.real()**

float imag(const float_complex& x)

Acquires the imaginary part. Return value: **x.imag()**

float abs(const float_complex& x)

Calculates the absolute value. Return value: $(|\mathbf{x}.real(\mathbf{j})|^2 + |\mathbf{x}.imag(\mathbf{j})|^2)^{1/2}$



float arg(const float_complex& x)

Calculates the phase angle. Return value: atan2f(x.imag(), x.real())

float norm(const float_complex& x)

Calculates the absolute value of the square. Return value: $|\mathbf{x}.real(\mathbf{j})|^2 + |\mathbf{x}.imag(\mathbf{j})|^2$

float_complex conj(const float_complex& x)

Calculates the conjugate complex number. Return value: float_complex(x.real(), (-1)*x.imag())

float_complex polar(const float& rho, const float& theta)

Calculates the **float_complex** value for a complex number with size **rho** and phase angle (argument) **theta**. Return value: **float_complex(rho*cosf(theta), rho*sinf(theta))**

float_complex cos(const float_complex& x)

Calculates the complex cosine.

Return value: float_complex(cosf(x.real())*coshf(x.imag()), (-1)*sinf(x.real())*sinhf(x.imag()))

float_complex cosh(const float_complex& x)

Calculates the complex hyperbolic cosine. Return value: cos(float_complex((-1)*x.imag(), x.real()))

float_complex& float_complex::operator-=(const float_complex& rhs)

Calculates the exponent function. Return value: expf(x.real())*cosf(x.imag()),expf(x.real())*sinf(x.imag())

float_complex log(const float_complex& x)

Calculates the natural logarithm (base e). Return value: **float_complex(logf(abs(x)), arg(x))**

float_complex log10(const float_complex& x)

Calculates the common logarithm (base 10). Return value: float_complex(log10f(abs(x)), arg(x)/logf(10))

float_complex pow(const float_complex& x, int y)

Calculates **x** to the **y**th power. If pow(0,0), a domain error will occur. Return value: If **float_complex pow(const float_complex& x, const float_complex& y)**: exp(y*logf(x)) Otherwise: exp(y*log(x))



float_complex pow(const float_complex& x, const float& y)

Calculates **x** to the **y**th power.

If pow(0,0), a domain error will occur.

Return value: If float_complex pow(const float_complex& x, const float_complex& y): exp(y*logf(x))

Otherwise: exp(y*log(x))

float_complex pow(const float_complex& x, const float_complex& y)

Calculates **x** to the **y**th power.

If pow(0,0), a domain error will occur.

Return value: If float_complex pow(const float_complex& x, const float_complex& y): exp(y*logf(x)) Otherwise: exp(y*log(x))

float_complex pow(const float& x, const float_complex& y)

Calculates **x** to the **y**th power.

If pow(0,0), a domain error will occur.

Return value: If float_complex pow(const float_complex& x, const float_complex& y): exp(y*logf(x)) Otherwise: exp(y*log(x))

float_complex sin(const float_complex& x)

Calculates the complex sine. Return value: float_complex(sinf(x.real())*coshf(x.imag()), cosf(x.real())*sinhf(x.imag()))

float_complex sinh(const float_complex& x)

Calculates the complex hyperbolic sine. Return value: float_complex(0,-1)*sin(float_complex((-1)*x.imag(),x.real()))

float_complex sqrt(const float_complex& x)

Calculates the square root within the right half space. Return value: float_complex(sqrtf(abs(x))*cosf(arg(x)/2), sqrtf(abs(x))*sinf(arg(x)/2))

float_complex tan(const float_complex& x)

Calculates the complex tangent. Return value: **sin(x)**/**cos(x)**

float_complex tanh(const float_complex& x)

Calculates the complex hyperbolic tangent. Return value: **sinh(x)/cosh(x)**



(c) double_complex Class

Туре	Definition Name	Description
Туре	value_type	double type
Variable	_re	Defines the real part of double precision
	_im	Defines the imaginary part of double precision
Function	double_complex(double re = 0.0, double im = 0.0)	Constructor
	double_complex(const float_complex&)	
	double real() const	Acquires the real part
	double imag() const	Acquires the imaginary part
	double_complex& operator=(double rhs)	Copies rhs to the real part 0.0 is assigned to the imaginary part
	double_complex& operator+=(double rhs)	Adds rhs to the real part of *this and stores the sum in *this
	double_complex& operator-=(double rhs)	Subtracts rhs from the real part of *this and stores the difference in *this .
	double_complex& operator*=(double rhs)	Multiplies * this by rhs and stores the product in * this
	double_complex& operator/=(double rhs)	Divides *this by rhs and stores the quotient in *this
	double_complex& operator=(const double_complex& rhs)	Copies rhs
	double_complex& operator+=(const double_complex& rhs)	Adds rhs to *this and stores the sum in *this
	double_complex& operator-=(const double_complex& rhs)	Subtracts rhs from *this and stores the difference in *this
	double_complex& operator*=(const double_complex& rhs)	Multiplies * this by rhs and stores the product in * this
	double_complex& operator/=(const double_complex& rhs)	Divides * this by rhs and stores the quotient in * this

double_complex::double_complex(double re = 0.0, double im = 0.0)

Constructor of class double_complex.

The initial settings are as follows:

_re = re;

_im = im;

double_complex::double_complex(const float_complex&)

Constructor of class **double_complex**. The initial settings are as follows:

- _re = (double)rhs.real();

_im = (double)rhs.imag();



double double_complex::real() const

Acquires the real part. Return value: **this->_re**

double double_complex::imag() const

Acquires the imaginary part. Return value: **this->_im**

double_complex& double_complex::operator=(double rhs)

Copies **rhs** to the real part (_**re**). 0.0 is assigned to the imaginary part (_**im**). Return value: ***this**

double_complex& double_complex::operator+=(double rhs)

Adds **rhs** to the real part (**_re**) and stores the result in the real part (**_re**). The value of the imaginary part (**_im**) does not change. Return value: ***this**

double_complex& double_complex::operator-= (double rhs)

Subtracts **rhs** from the real part (**_re**) and stores the result in the real part (**_re**). The value of the imaginary part (**_im**) does not change. Return value: ***this**

double_complex& double_complex::operator*=(double rhs)

Multiplies *this by rhs and stores the result in *this. (_re=_re*rhs, _im=_im*rhs) Return value: *this

double_complex& double_complex::operator/=(double rhs)

Divides ***this** by **rhs** and stores the result in ***this**. (_re=_re/rhs, _im=_im/rhs) Return value: ***this**

double_complex& double_complex::operator=(const double_complex& rhs)

Copies **rhs** to ***this**. Return value: ***this**

double_complex& double_complex::operator+=(const double_complex& rhs)

Adds **rhs** to ***this** and stores the result in ***this**. Return value: ***this**

double_complex& double_complex::operator-=(const double_complex& rhs)

Subtracts **rhs** from ***this** and stores the result in ***this**. Return value: ***this**



double_complex& double_complex::operator*=(const double_complex& rhs)

Multiplies ***this** by **rhs** and stores the result in ***this**. Return value: ***this**

double_complex& double_complex::operator/=(const double_complex& rhs)

Divides ***this** by **rhs** and stores the result in ***this**. Return value: ***this**

(d) double_complex Non-Member Function

Туре	Definition Name	Description
Function	double_complex operator+(const double_complex& lhs)	Performs unary + operation of Ihs
	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)	



Туре	Definition Name	Description
	double_complex operator/(const double_complex& lhs, const double_complex& rhs)	Returns the result of dividing Ihs by rhs
	double_complex operator/(const double_complex& lhs, const double& rhs)	
	double_complex operator/(const double& lhs, const double_complex& rhs)	
Function	bool operator==(const double_complex& lhs, const double_complex& rhs)	Compares the real part of lhs and rhs , and the imaginary parts of lhs and rhs
	bool operator==(const double_complex& lhs, const double& rhs)	
	bool operator==(const double& lhs, const double_complex& rhs)	
	bool operator!=(const double_complex& lhs, const double_complex& rhs)	Compares the real parts of lhs and rhs , and the imaginary parts of lhs and rhs
	bool operator!=(const double_complex& lhs, const double& rhs)	
	bool operator!=(const double& lhs, const double_complex& rhs)	
	istream& operator>>(istream& is, double_complex& x)	Inputs x in a format of u , (u), or (u , v) (u : real part, v : imaginary part)
	ostream& operator<<(ostream& os, const double_complex& x)	Outputs x in a format of u , (u), or (u , v) (u : real part, v : imaginary part)
	double real(const double_complex& x)	Acquires the real part
	double imag(const double_complex& x)	Acquires the imaginary part
	double abs(const double_complex& x)	Calculates the absolute value
	double arg(const double_complex& x)	Calculates the phase angle
	double norm(const double_complex& x)	Calculates the absolute value of the square
	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



Туре	Definition Name	Description
	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
Function	double_complex pow(const double_complex& x, int y)	Calculates x to the y th power
	double_complex pow(const double_complex& x, const double& y)	
	double_complex pow(const double_complex& x, const double_complex& y)	
	double_complex pow(const double& x, const double_complex& y)	
	double_complex sin(const double_complex& x)	Calculates the complex sine
	double_complex sinh(const double_complex& x)	Calculates the complex hyperbolic sine
	double_complex sqrt(const double_complex& x)	Calculates the square root within the right half space
	double_complex tan(const double_complex& x)	Calculates the complex tangent
	double_complex tanh(const double_complex& x)	Calculates the complex hyperbolic tangent

double_complex operator+(const double_complex& lhs)

Performs unary + operation of **lhs**. Return value: **lhs**

double_complex operator+(const double_complex& lhs, const double_complex& rhs)

Returns the result of adding **lhs** to **rhs**. Return value: **double_complex(lhs)+=rhs**

double_complex operator+(const double_complex& lhs, const double& rhs)

Returns the result of adding **lhs** to **rhs**. Return value: **double_complex(lhs)+=rhs**

double_complex operator+(const double& lhs, const double_complex& rhs)

Returns the result of adding **lhs** to **rhs**. Return value: **double_complex(lhs)+=rhs**



double_complex operator-(const double_complex& lhs)

Performs unary - operation of Ihs. Return value: double_complex(-lhs.real(), -lhs.imag())

double_complex operator-(const double_complex& lhs, const double_complex& rhs)

Returns the result of subtracting **rhs** from **lhs**. Return value: **double_complex(lhs)-=rhs**

double_complex operator-(const double_complex& lhs, const double& rhs)

Returns the result of subtracting **rhs** from **lhs**. Return value: **double_complex(lhs)-=rhs**

double_complex operator-(const double& lhs, const double_complex& rhs)

Returns the result of subtracting **rhs** from **lhs**. Return value: **double_complex(lhs)-=rhs**

double_complex operator*(const double_complex& lhs, const double_complex& rhs)

Returns the result of multiplying **lhs** by **rhs**. Return value: **double_complex(lhs)*=rhs**

double_complex operator*(const double_complex& lhs, const double& rhs)

Returns the result of multiplying **lhs** by **rhs**. Return value: **double_complex(lhs)*=rhs**

double_complex operator*(const double& lhs, const double_complex& rhs)

Returns the result of multiplying **lhs** by **rhs**. Return value: **double_complex(lhs)*=rhs**

double_complex operator/(const double_complex& lhs, const double_complex& rhs)

Returns the result of dividing **lhs** by **rhs**. Return value: **double_complex(lhs)**/=**rhs**

double_complex operator/(const double_complex& lhs, const double& rhs)

Returns the result of dividing **lhs** by **rhs**. Return value: **double_complex(lhs)/=rhs**

double_complex operator/(const double& lhs, const double_complex& rhs)

Returns the result of dividing **lhs** by **rhs**. Return value: **double_complex(lhs)**/=**rhs**

bool operator==(const double_complex& lhs, const double_complex& rhs)

Compares the real parts of Ihs and rhs, and the imaginary parts of Ihs and rhs.



For a **double** type parameter, the imaginary part is assumed to be 0.0. Return value: lhs.real()==rhs.real() && lhs.imag()==rhs.imag()

bool operator==(const double_complex& lhs, const double& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **double** type parameter, the imaginary part is assumed to be 0.0. Return value: Ihs.real()==rhs.real() && Ihs.imag()==rhs.imag()

bool operator==(const double& lhs, const double_complex& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **double** type parameter, the imaginary part is assumed to be 0.0. Return value: Ihs.real()==rhs.real() && Ihs.imag()==rhs.imag()

bool operator!=(const double_complex& lhs, const double_complex& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **double** type parameter, the imaginary part is assumed to be 0.0. Return value: **Ihs.real()**!=**rhs.real()** || **Ihs.imag()**!=**rhs.imag()**

bool operator!=(const double_complex& lhs, const double& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **double** type parameter, the imaginary part is assumed to be 0.0. Return value: **Ihs.real()**!=**rhs.real()** || **Ihs.imag()**!=**rhs.imag()**

bool operator!=(const double& lhs, const double_complex& rhs)

Compares the real parts of **Ihs** and **rhs**, and the imaginary parts of **Ihs** and **rhs**. For a **double** type parameter, the imaginary part is assumed to be 0.0. Return value: **Ihs.real()**!=**rhs.real()** || **Ihs.imag()**!=**rhs.imag()**

istream& operator>>(istream& is, double_complex& x)

Inputs complex number **x** in a format of **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part). The input value is converted to **double_complex**. If **x** is input in a format other than the **u**, (**u**), or (**u**,**v**) format, **is.setstate(ios_base::failbit)** is called. Return value: **is**

ostream& operator<<(ostream& os, const double_complex& x)

Outputs **x** to **os**. The output format is **u**, (**u**), or (**u**,**v**) (**u**: real part, **v**: imaginary part). Return value: **os**

double real(const double_complex& x)

Acquires the real part. Return value: **x.real()**

double imag(const double_complex& x)

Acquires the imaginary part. Return value: **x.imag()**



double abs(const double_complex& x)

Calculates the absolute value. Return value: $(|\mathbf{x}.real()|^2 + |\mathbf{x}.imag()|^2)^{1/2}$

double arg(const double_complex& x)

Calculates the phase angle. Return value: atan2(x.imag(), x.real())

double norm(const double_complex& x)

Calculates the absolute value of the square. Return value: $|\mathbf{x}.real()|^2 + |\mathbf{x}.imag()|^2$

double_complex conj(const double_complex& x)

Calculates the conjugate complex number. Return value: double_complex(x.real(), (-1)*x.imag())

double_complex polar(const double& rho, const double& theta)

Calculates the **double_complex** value for a complex number with size **rho** and phase angle (argument) **theta**. Return value: **double_complex(rho*cos(theta), rho*sin(theta))**

double_complex cos(const double_complex& x)

Calculates the complex cosine.

Return value: double_complex(cos(x.real())*cosh(x.imag()), (-1)*sin(x.real())*sinh(x.imag()))

double_complex cosh(const double_complex& x)

Calculates the complex hyperbolic cosine. Return value: cos(double_complex((-1)*x.imag(), x.real()))

double_complex exp(const double_complex& x)

Calculates the exponent function. Return value: exp(x.real())*cos(x.imag()),exp(x.real())*sin(x.imag())

double_complex log(const double_complex& x)

Calculates the natural logarithm (base e). Return value: double_complex(log(abs(x)), arg(x))

double_complex log10(const double_complex& x)

Calculates the common logarithm (base 10). Return value: double_complex(log10(abs(x)), arg(x)/log(10))



double_complex pow(const double_complex& x, int y)

Calculates **x** to the **y**th power. If pow(0,0), a domain error will occur. Return value: $exp(y^*log(x))$

double_complex pow(const double_complex& x, const double& y)

Calculates **x** to the **y**th power. If pow(0,0), a domain error will occur. Return value: exp(y*log(x))

double_complex pow(const double_complex& x, const double_complex& y)

Calculates **x** to the **y**th power. If pow(0,0), a domain error will occur. Return value: exp(y*log(x))

double_complex pow(const double& x, const double_complex& y)

Calculates **x** to the **y**th power. If pow(0,0), a domain error will occur. Return value: exp(y*log(x))

double_complex sin(const double_complex& x)

Calculates the complex sine Return value: double_complex(sin(x.real())*cosh(x.imag()), cos(x.real())*sinh(x.imag()))

double_complex sinh(const double_complex& x)

Calculates the complex hyperbolic sine Return value: double_complex(0,-1)*sin(double_complex((-1)*x.imag(),x.real()))

double_complex sqrt(const double_complex& x)

Calculates the square root within the right half space Return value: double_complex(sqrt(abs(x))*cos(arg(x)/2), sqrt(abs(x))*sin(arg(x)/2))

double_complex tan(const double_complex& x)

Calculates the complex tangent. Return value: **sin(x)/cos(x)**

double_complex tanh(const double_complex& x)

Calculates the complex hyperbolic tangent. Return value: **sinh(x)/cosh(x)**



6.5.4 String Handling Class Library

The header file for the string handling class library is as follows:

- <string>

Defines class string.

This class has no derivation.

(a) string Class

Туре	Definition Name	Description
Туре	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



Туре	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)	



Туре	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 ${\bf c}$ before the string specified by ${\bf p}$



Туре	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
	<pre>string& erase(size_t pos = 0, size_t n = npos)</pre>	Deletes n characters from position pos
	iterator erase(iterator position)	Deletes the character referenced by posi- tion
	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 start- ing 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 start- ing 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 start- ing 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 start- ing from position pos with n2 characters, each of which is c



Туре	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 posi- tion pos
	void swap(string& str)	Swaps *this with string str
	const char* c_str() const	References the pointer to the memory area
	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



Туре	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



Туре	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
	int compare(const char* str) const	Compares *this with string str
	int compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const	Compares the string of n1 characters from position pos1 with n2 characters of string str

string::string(void)

Sets as follows: s_ptr = 0;



s_len = 0; s_res = 1;

string::string(const string& str, size_t pos = 0, size_t n = npos)

Copies str. Note that s_len will be the smaller value of n and s_len.

string::string(const char* str, size_t n)

Sets as follows: s_ptr = **str**; s_len = **n**; s_res = **n** + 1;

string::string(const char* str)

Sets as follows: s_ptr = str; s_len = length of string str; s_res = length of string str + 1;

string::string(size_t n, char c)

Sets as follows: $s_ptr = string of n$ characters, each of which is c $s_len = n;$ $s_res = n + 1;$

string::~string()

Destructor of class **string**. Deallocates the memory area where the string is stored.

string& string::operator(const string& str)

Assigns the data of **str**. Return value: ***this**

string& string::operator=(const char* str)

Creates a **string** object from **str** and assigns its data to the **string** object. Return value: ***this**

string& string::operator=(char c)

Creates a $\ensuremath{\textit{string}}$ object from $\ensuremath{\textit{c}}$ and assigns its data to the $\ensuremath{\textit{string}}$ object. Return value: *this

string::iterator string::begin()

Calculates the start pointer of the string. Return value: Start pointer of the string



string::const_iterator string::begin() const

Calculates the start pointer of the string. Return value: Start pointer of the string

string::iterator string::end()

Calculates the end pointer of the string. Return value: End pointer of the string

string::const_iterator string::end() const

Calculates the end pointer of the string. Return value: End pointer of the string

size_t string::size() const

Calculates the length of the stored string. Return value: Length of the stored string

size_t string::length() const

Calculates the length of the stored string. Return value: Length of the stored string

size_t string::max_size() const

Calculates the size of the allocated memory area. Return value: Size of the allocated area

void string::resize(size_t n, char c)

Changes the number of characters in the string that can be stored by the object to **n**.

If **n**<=**size()**, replaces the string with the original string with length **n**.

If n>size(), replaces the string with a string that has c appended to the end so that the length will be equal to n.

The length must be **n<=max_size()**.

If n>max_size(), the string length is n=max_size().

void string::resize(size_t n)

Changes the number of characters in the string that can be stored by the object to **n**. If **n**<=**size()**, replaces the string with the original string with length **n**. The length must be **n**<=**max_size**.

size_t string::capacity() const

Calculates the size of the allocated memory area. Return value: Size of the allocated memory area

void string::reserve(size_t res_arg = 0)

Re-allocates the memory area.

After reserve(), capacity() will be equal to or larger than the reserve() parameter.

When the memory area is re-allocated, all references, pointers, and **iterator** that references the elements of the numeric sequence become invalid.



void string::clear()

Clears the stored string.

bool string::empty() const

Checks whether the number of characters in the stored string is 0. Return value: If the length of the stored string is 0: **true** If the length of the stored string is not zero: **false**

const char& string::operator[](size_t pos) const

References s_ptr[pos]. Return value: If n< s_len: s_ptr [pos] If n>= s_len: '\0'

char& string::operator[](size_t pos)

References **s_ptr[pos]**. Return value: If **n< s_len**: **s_ptr [pos]** If **n>= s_len**: '\0'

const char& string::at(size_t pos) const

References s_ptr[pos]. Return value: If n< s_len: s_ptr [pos] If n>= s_len: '\0'

char& string::at(size_t pos)

References s_ptr[pos]. Return value: If n< s_len: s_ptr [pos] If n>= s_len: '\0'

string& string::operator+=(const string& str)

Appends the string stored in **str** to the object. Return value: ***this**

string& string::operator+=(const char* str)

Creates a **string** object from **str** and adds the string to the object. Return value: ***this**

string& string::operator+=(char c)

Creates a **string** object from **c** and adds the string to the object. Return value: ***this**

string& string::append(const string& str)

Appends string **str** to the object. Return value: ***this**



string& string::append(const char* str)

Appends string **str** to the object. Return value: ***this**

string& string::append(const string& str, size_t pos, size_t n);

Appends **n** characters of string **str** to the object position **pos**. Return value: ***this**

string& string::append(const char* str, size_t n)

Appends **n** characters of string **str** to the object. Return value: ***this**

string& string::append(size_t n, char c)

Appends **n** characters, each of which is \mathbf{c} , to the object. Return value: ***this**

string& string::assign(const string& str)

Assigns string **str**. Return value: ***this**

string& string::assign(const char* str)

Assigns string **str**. Return value: ***this**

string& string::assign(const string& str, size_t pos, size_t n)

Assigns **n** characters of string **str** to position **pos**. Return value: ***this**

string& string::assign(const char* str, size_t n)

Assigns **n** characters of string **str**. Return value: ***this**

string& string::assign(size_t n, char c)

Assigns **n** characters, each of which is **c**. Return value: ***this**

string& string::insert(size_t pos1, const string& str)

Inserts string **str** to position **pos1**. Return value: ***this**

string& string::insert(size_t pos1, const string& str, size_t pos2, size_t n)

Inserts n characters starting from position pos2 of string str to position pos1.



Return value: *this

string& string::insert(size_t pos, const char* str, size_t n)

Inserts **n** characters of string **str** to position **pos**. Return value: ***this**

string& string::insert(size_t pos, const char* str)

Inserts string **str** to position **pos**. Return value: ***this**

string& string::insert(size_t pos, size_t n, char c)

Inserts a string of **n** characters, each of which is **c**, to position **pos**. Return value: ***this**

string::iterator string::insert(iterator p, char c = char())

Inserts character **c** before the string specified by **p**. Return value: The inserted character

void string::insert(iterator p, size_t n, char c)

Inserts n characters, each of which is c, before the character specified by p.

string& string::erase(size_t pos = 0, size_t n = npos)

Deletes **n** characters starting from position **pos**. Return value: ***this**

iterator string::erase(iterator position)

Deletes the character referenced by position.

Return value: If the next iterator of the element to be deleted exists: The next iterator of the deleted element If the next iterator of the element to be deleted does not exist: end()

iterator string::erase(iterator first, iterator last)

Deletes the characters in range [first, last]. Return value: If the next iterator of last exists: The next iterator of last If the next iterator of last does not exist: end()

string& string::replace(size_t pos1, size_t n1, const string& str)

Replaces the string of **n1** characters starting from position **pos1** with string **str**. Return value: ***this**

string& string::replace(size_t pos1, size_t n1, const char* str)

Replaces the string of **n1** characters starting from position **pos1** with string **str**. Return value: ***this**



string& string::replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)

Replaces the string of **n1** characters starting from position **pos1** with the string of **n2** characters starting from position **pos2** in string **str**.

Return value: *this

string& string::replace(size_t pos, size_t n1, const char* str, size_t n2)

Replaces the string of **n1** characters starting from position **pos1** with **n2** characters of string **str**. Return value: ***this**

string& string::replace(size_t pos, size_t n1, size_t n2, char c)

Replaces the string of **n1** characters starting from position **pos** with **n2** characters, each of which is **c**. Return value: ***this**

string& string::replace(iterator i1, iterator i2, const string& str)

Replaces the string from position **i1** to **i2** with string **str**. Return value: ***this**

string& string::replace(iterator i1, iterator i2, const char* str)

Replaces the string from position **i1** to **i2** with string **str**. Return value: ***this**

string& string::replace(iterator i1, iterator i2, const char* str, size_t n)

Replaces the string from position i1 to i2 with n characters of string str Return value: *this

string& string::replace(iterator i1, iterator i2, size_t n, char c)

Replaces the string from position i1 to i2 with n characters, each of which is ${\mbox{c}}.$ Return value: *this

size_t string::copy(char* str, size_t n, size_t pos = 0) const

Copies **n** characters of string **str** to position **pos**. Return value: **rlen**

void string::swap(string& str)

Swaps *this with string str.

const char* string::c_str() const

References the pointer to the memory area where the string is stored. Return value: **s_ptr**

const char* string::data() const

References the pointer to the memory area where the string is stored.


Return value: s_ptr

size_t string::find(const string& str, size_t pos = 0) const

Finds the position where the string same as string **str** first appears after position **pos**. Return value: Offset of string

size_t string::find (const char* str, size_t pos = 0) const

Finds the position where the string same as string **str** first appears after position **pos**. Return value: Offset of string

size_t string::find(const char* str, size_t pos, size_t n) const

Finds the position where the string same as **n** characters of string **str** first appears after position **pos**. Return value: Offset of string

size_t string::find(char c, size_t pos = 0) const

Finds the position where character c first appears after position **pos**. Return value: Offset of string

size_t string::rfind(const string& str, size_t pos = npos) const

Finds the position where a string same as string **str** appears most recently before position **pos**. Return value: Offset of string

size_t string::rfind(const char* str, size_t pos = npos) const

Finds the position where a string same as string **str** appears most recently before position **pos**. Return value: Offset of string

size_t string::rfind(const char* str, size_t pos, size_t n) const

Finds the position where the string same as **n** characters of string **str** appears most recently before position **pos**. Return value: Offset of string

size_t string::rfind(char c, size_t pos = npos) const

Finds the position where character \mathbf{c} appears most recently before position \mathbf{pos} . Return value: Offset of string

size_t string::find_first_of(const string& str, size_t pos = 0) const

Finds the position where any character included in string **str** first appears after position **pos**. Return value: Offset of string

size_t string::find_first_of(const char* str, size_t pos = 0) const

Finds the position where any character included in string **str** first appears after position **pos**. Return value: Offset of string



size_t string::find_first_of(const char* str, size_t pos, size_t n) const

Finds the position where any character included in **n** characters of string **str** first appears after position **pos**. Return value: Offset of string

size_t string::find_first_of(char c, size_t pos = 0) const

Finds the position where character c first appears after position **pos**. Return value: Offset of string

size_t string::find_last_of(const string& str, size_t pos = npos) const

Finds the position where any character included in string **str** appears most recently before position **pos**. Return value: Offset of string

size_t string::find_last_of(const char* str, size_t pos = npos) const

Finds the position where any character included in string **str** appears most recently before position **pos**. Return value: Offset of string

size_t string::find_last_of(const char* str, size_t pos, size_t n) const

Finds the position where any character included in **n** characters of string **str** appears most recently before position **pos**. Return value: Offset of string

size_t string::find_last_of(char c, size_t pos = npos) const

Finds the position where character **c** appears most recently before position **pos**. Return value: Offset of string

size_t string::find_first_not_of(const string& str, size_t pos = 0) const

Finds the position where a character different from any character included in string **str** first appears after position **pos**. Return value: Offset of string

size_t string::find_first_not_of(const char* str, size_t pos = 0) const

Finds the position where a character different from any character included in string **str** first appears after position **pos**. Return value: Offset of string

size_t string::find_first_not_of(const char* str, size_t pos, size_t n) const

Finds the position where a character different from any character in the first **n** characters of string **str** first appears after position **pos**.

Return value: Offset of string

size_t string::find_first_not_of(char c, size_t pos = 0) const

Finds the position where a character different from character c first appears after position **pos**. Return value: Offset of string



size_t string::find_last_not_of(const string& str, size_t pos = npos) const

Finds the position where a character different from any character included in string **str** appears most recently before position **pos**.

Return value: Offset of string

size_t string::find_last_not_of(const char* str, size_t pos = npos) const

Finds the position where a character different from any character included in string **str** appears most recently before position **pos**.

Return value: Offset of string

size_t string::find_last_not_of(const char* str, size_t pos, size_t n) const

Finds the position where a character different from any character in the first **n** characters of string **str** appears most recently before position **pos**.

Return value: Offset of string

size_t string::find_last_not_of(char c, size_t pos = npos) const

Finds the position where a character different from character **c** appears most recently before position **pos**. Return value: Offset of string

string string::substr(size_t pos = 0, size_t n = npos) const

Creates an object from a string in the range [**pos**,**n**] of the stored string. Return value: Object with a string in the range [**pos**,**n**]

int string::compare(const string& str) const

Compares the string with string str.

Return value: If the strings are the same: 0 If the strings are different: 1 when this->s_len > str.s_len, -1 when this->s_len < str.s_len

int string::compare(size_t pos1, size_t n1, const string& str) const

Compares a string of **n1** characters starting from position **pos1** of *this with string str. Return value: If the strings are the same: 0 If the strings are different: 1 when this->s_len > str.s_len, -1 when this->s_len < str.s_len

int string::compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const

Compares a string of **n1** characters starting from position **pos1** with the string of **n2** characters from position **pos2** of string **str**.

Return value: If the strings are the same: 0

If the strings are different: 1 when this->s_len > str.s_len,

-1 when this->s_len < str.s_len

int string::compare(const char* str) const

Compares *this with string str.

Return value: If the strings are the same: 0

If the strings are different: 1 when this->s_len > str.s_len,

-1 when this->s_len < str.s_len



int string::compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const

Compares the string of **n1** characters from position **pos1** with **n2** characters of string **str**.

Return value: If the strings are the same: 0

If the strings are different: 1 when this->s_len > str.s_len, -1 when this->s_len < str.s_len

(b) string Class Manipulators

Туре	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
	string operator+(const char* lhs, const string& rhs)	in the object
	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 Ihs 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 Ihs with the
	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 Ihs with the string length of rhs
	bool operator<(const char* lhs, const string& rhs)	Compares the string length of lhs with
	bool operator<(const string& lhs, const char* rhs)	the string length of rns
	bool operator>(const string& lhs, const string& rhs)	Compares the string length of lhs with
	bool operator>(const char* lhs, const string& rhs)	the string length of rns
	bool operator>(const string& lhs, const char* rhs)	
	bool operator<=(const string& lhs, const string& rhs)	Compares the string length of Ihs with the string length of rhs
	bool operator<=(const char* lhs, const string& rhs)	
	bool operator<=(const string& lhs, const char* rhs)	



Туре	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 Ihs with the string of rhs
	istream& operator>>(istream& is, string& str)	Extracts the string to str
	ostream& operator<<(ostream& os, const string& str)	Inserts string str
	istream& getline(istream& is, string& str, char delim)	Extracts a string from is and appends it to str . If delim is found in the string, input is stopped.
	istream& getline(istream& is, string& str)	Extracts a string from is and appends it to str . If a new-line character is detected, input is stopped.

string operator+(const string& lhs, const string& rhs)

Appends the string (characters) of **Ihs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.

Return value: Object where the linked strings are stored

string operator+(const char* lhs, const string& rhs)

Appends the string (characters) of **Ihs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.

Return value: Object where the linked strings are stored

string operator+(char lhs, const string& rhs)

Appends the string (characters) of **Ihs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.

Return value: Object where the linked strings are stored

string operator+(const string& lhs, const char* rhs)

Appends the string (characters) of **Ihs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.

Return value: Object where the linked strings are stored

string operator+(const string& lhs, char rhs)

Appends the string (characters) of **lhs** with the strings (characters) of **rhs**, creates an object and stores the string in the object.

Return value: Object where the linked strings are stored

bool operator==(const string& lhs, const string& rhs)

Compares the string of Ihs with the string of rhs.



Return value: If the strings are the same: true If the strings are different: false

bool operator==(const char* lhs, const string& rhs)

Compares the string of **Ihs** with the string of **rhs**. Return value: If the strings are the same: **true** If the strings are different: **false**

bool operator==(const string& lhs, const char* rhs)

Compares the string of **Ihs** with the string of **rhs**. Return value: If the strings are the same: **true** If the strings are different: **false**

bool operator!=(const string& lhs, const string& rhs)

Compares the string of **Ihs** with the string of **rhs**. Return value: If the strings are the same: **false**

bool operator!=(const char* lhs, const string& rhs)

Compares the string of **Ihs** with the string of **rhs**. Return value: If the strings are the same: **false**

bool operator!=(const string& lhs, const char* rhs)

Compares the string of **Ihs** with the string of **rhs**. Return value: If the strings are the same: **false** If the strings are different: **true**

bool operator<(const string& lhs, const string& rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len < rhs.s_len**: **true** If **Ihs.s_len >= rhs.s_len**: **false**

bool operator<(const char* lhs, const string& rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len** < **rhs.s_len**: **true** If **Ihs.s_len** >= **rhs.s_len**: **false**

bool operator<(const string& lhs, const char* rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len < rhs.s_len**: **true** If **Ihs.s_len >= rhs.s_len**: **false**

bool operator>(const string& lhs, const string& rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len** > **rhs.s_len**: **true**



bool operator>(const char* lhs, const string& rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len** > **rhs.s_len**: **true**

bool operator>(const string& lhs, const char* rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len** > **rhs.s_len**: **true** If **Ihs.s_len** <= **rhs.s_len**: **false**

bool operator<=(const string& lhs, const string& rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len** <= **rhs.s_len**: **true** If **Ihs.s_len** > **rhs.s_len**: **false**

bool operator<=(const char* lhs, const string& rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len <= rhs.s_len**: **true** If **Ihs.s_len > rhs.s_len**: **false**

bool operator<=(const string& lhs, const char* rhs)</pre>

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len** <= **rhs.s_len**: **true** If **Ihs.s_len** > **rhs.s_len**: **false**

bool operator>=(const string& lhs, const string& rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len** >= **rhs.s_len**: **true** If **Ihs.s_len** < **rhs.s_len**: **false**

bool operator>=(const char* lhs, const string& rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len** >= **rhs.s_len**: **true** If **Ihs.s_len** < **rhs.s_len**: **false**

bool operator>=(const string& lhs, const char* rhs)

Compares the string length of **Ihs** with the string length of **rhs**. Return value: If **Ihs.s_len** >= **rhs.s_len**: **true** If **Ihs.s_len** < **rhs.s_len**: **false**

void swap(string& lhs,string& rhs)

Swaps the string of $\boldsymbol{\mathsf{lhs}}$ with the string of $\boldsymbol{\mathsf{rhs}}.$

istream& operator>>(istream& is, string& str)

Extracts a string to str.



Return value: is

ostream& operator<<(ostream& os, const string& str)

Inserts string **str**. Return value: **os**

istream& getline(istream& is, string& str, char delim)

Extracts a string from **is** and appends it to **str**. If **delim** is found in the string, the input is stopped. Return value: **is**

istream& getline(istream& is, string& str)

Extracts a string from **is** and appends it to **str**. If a new-line character is found, the input is stopped. Return value: **is**

6.6 Unsupported Libraries

Table 6.15 lists the libraries which are specified in the C language specifications but not supported by this compiler.

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, wcs- tombs
5	string.h	strcoll, strxfrm
6	time.h	clock, difftime, mktime, time, asctime, ctime, gmtime, localtime, strftime
7	wctype.h	iswalnum, iswalpha, iswblank, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, iswctype, wctype, towlower, towupper, towctrans, wctrans
8	wchar.h	wcsftime, wcscoll, wcsxfrm, wctob, mbrtowc, wcrtomb, mbsrtowcs, wcsrtombs

Table 6.15 Unsupported Libraries

Note

The header file is not supported.



7. STARTUP

This chapter describes the startup routine.

7.1 Overview

To execute a C language program, a program is needed to handle ROMization for inclusion in the system and to start the user program (main function). This program is called the startup routine.

To execute a user program, a startup routine must be created for that program. The CubeSuite+ provides standard startup routine object files, which carry out the processing required before program execution, and the startup routine source files (assembly source), which the user can adapt to the system.

7.2 File Contents

Startup routine that CubeSuite+ supplies is as follows:

	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)
(I)	sbrk.h	Low-level interface routine (sbrk function header)

Table 7.1 List of Programs Created in Integrated Development Environment

7.3 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 7.5.7 Application Startup.

A summary of the necessary procedures is given below.

- Fixed vector table setting

Sets the fixed vector table to initiate the initial setting routine (**PowerOn_Reset_PC**) at a power-on reset. In addition to the reset vector, processing routines, such as, privileged instruction exception, access exception, undefined instruction exception, floating-point exception, and nonmaskable interrupt, can be registered to the fixed vector table.

- Initial setting

Performs the procedures required to reach the **main** function. Registers and sections are initialized and various initial setting routines are called.

- Low-level interface routine creation

Routines providing an interface between the user system and library functions which are necessary when standard I/ O (stdio.h, ios, streambuf, istream, and ostream) and memory management libraries (stdlib.h and new) are used.

- Termination processing routine (exit, atexit, and abort)* creation Processing for terminating the program is performed.
- Note

* When using the C library function exit, atexit, or abort to terminate a program, these functions must be created as appropriate to the user system. When using the C++ program or C library macro assert, the abort function must always be created.

7.3.1 Fixed Vector Table Setting

To call the initial setting routine (PowerOn_Reset_PC) at a power-on reset, set the address of PowerOn_Reset_PC to the reset vector of the fixed vector table. A coding example is shown below.

In addition to the reset vector, processing routines, such as, privileged instruction exception, access exception, undefined instruction exception, floating-point exception, and nonmaskable interrupt, can be registered to the fixed vector table. For details on the fixed vector table, refer to the hardware manual.

Example:

```
extern void PowerOn_Reset_PC(void);
#pragma section C FIXEDVECT
                               /* Outputs RESET_Vectors to the FIXEDVECT*/
                               /* section by #pragma section declaration.*/
                               /* Allocates the FIXEDVECT section to reset*/
                               /* vector by the start option at linkage.*/
void (*const RESET_Vectors[])(void)={
    PowerOn_Reset_PC,
};
```

7.3.2 Initial Setting

The initial setting routine (PowerOn Reset PC) is a function that contains the procedures required before and after executing the main function. Processings required in the initial setting routine are described below in order.

- Initialization of PSW for Initial Setting Processing (1)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).
- Initialization of Stack Pointer (2)

The stack pointer (USP register and ISP register) is initialized. The #pragma entry declaration for the PowerOn_Reset_PC function makes the compiler automatically create the ISP/USP initialization code at the beginning of the function. This procedure does not have to be written because the PowerOn_Reset_PC function is declared by #pragma

entry.

- Initialization of General Registers Used as Base Registers (3) When the **base** option is used in the compiler, general registers used as base addresses in the entire program need to be initialized. The #pragma entry declaration for the PowerOn_Reset_PC function makes the compiler automatically create the initialization code for each register at the beginning of the function. This procedure does not have to be written because the PowerOn Reset PC function is declared by #pragma entry.
- Initialization of Control Registers (4)

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.

Initialization Processing of Sections (5)

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;	<pre>//Start address member of the initialized data //section in ROM</pre>
<pre>void *rom_e;</pre>	//End address member of the initialized data
	//section in ROM
void *ram_s;	//Start address member of the initialized data
	//section in RAM
<pre>} DTBL[] = {sectop("D"),</pre>	secend("D"),sectop("R")};
<pre>#pragma section C C\$BSEC extern const struct {</pre>	//Section name must be C\$BSEC
void *b_s;	//Start address member of the uninitialized data section
void *b_e;	//End address member of the uninitialized data section
<pre>} BTBL[] = {sectop("B"),</pre>	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 *_slptr;
#ifdef __cplusplus
extern "C" {
#endif
void _INITLIB (void)
{
                                   // Set initial setting for low-level
    _INIT_LOWLEVEL();
                                   // 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)
ł
                                   // Set initial setting if using rand function
    srand(1);
                                   // Set initial setting if using strtok function
   _slptr=NULL;
}
#ifdef __cplusplus
}
#endif
```



Notes 1. Specify the filename for the standard I/O file. This name is used in the low-level interface routine "open".

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

- (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 3.2.6 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.

7.3.3 Coding Example of Initial Setting Routine

A coding example of the **PowerOn_Reset_PC** function is shown here. For the actual initial setting routine created in the integrated development environment, refer to 7.4 Coding Example.



```
#include <machine.h>
#include <_h_c_lib.h>
#include "typedefine.h"
#include "stacksct.h"
#ifdef __cplusplus
extern "C" {
#endif
void PowerOn_Reset_PC(void);
void main(void);
#ifdef __cplusplus
}
#endif
                                               // Use SIM I/O
#ifdef __cplusplus
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_PC
void PowerOn_Reset_PC(void)
{
set_intb(__sectop("C$VECT"));
set_fpsw(FPSW_init);
_INITSCT();
_INITLIB();
    nop();
set_psw(PSW_init);
main();
brk();
}
```

7.3.4 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 7.2 lists the low-level interface routines used by C library functions.

Table 7.2 List of Low-Level Interface Routine

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

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 (**Iseek** 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)

[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) (The name of the parameter appearing in the interface) (Meaning) (The value passed as a parameter)



long open (const char *name, long mode, long flg)

[Description]

- Prepares for operations on the file corresponding to the filename of the first parameter. In the open routine, the file type (console, printer, disk file, etc.) must be determined in order to enable writing or reading at a later time. The file type must be referenced using the file number returned by the open routine each time reading or writing is to be performed.
- The second parameter mode specifies processing to be performed when the file is opened. The meanings of each of the bits of this parameter are as follows.



Table 7.3	Explanation of Bits	in Parameter	"mode" of	open Routine

mode Bit	Description
O_RDONLY (bit 0)	When this bit is 1, the file is opened in read-only mode
O_WRONLY (bit 1)	When this bit is 1, the file is opened in write-only mode
O_RDWR (bit 2)	When this bit is 1, the file is opened for both reading and writing
O_CREAT (bit 3)	When this bit is 1, if a file with the filename given does not exist, it is created
O_TRUNC (bit 4)	When this bit is 1, if a file with the filename given exists, the file contents are deleted and the file size is set to 0
O_APPEND (bit 5)	Sets the position within the file for the next read/write operation When 0: Set to read/write from the beginning of file When 1: Set to read/write from file end

- When there is a contradiction between the file processing specified by mode and the properties of the actual file, error processing should be performed. When the file is opened normally, the file number (a positive integer) should be returned which should be used in subsequent read, write, Iseek, 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: Error:

The file number for the successfully opened file -1

[Parameters]

flg

name	Name of the file
mode	Specifies the type of processing when the file is opened
flg	Specifies processing when the file is opened (always 0777)



long close (long fileno)

[Description]

- The file number obtained using the open routine is passed as a parameter.
- The file management information area set using the open routine should be released to enable reuse. Also, when output file buffering is performed in low-level interface routines, the buffer contents should be written to the actual file.
- When the file is closed successfully, 0 is returned; if the close operation fails, -1 is returned.

[Return value]

Normal:	0
Error:	-1

[Parameters]

fileno

Number of the file to be closed



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

[Description]

- Data is read from the file specified by the first parameter (fileno) to the area in memory specified by the second parameter (buf). The number of bytes of data to be read is specified by the third parameter (count).
- When the end of the file is reached, only a number of bytes fewer than or equal to count bytes can be read.
- The position for file reading/writing advances by the number of bytes read.
- When reading is performed successfully, the actual number of bytes read is returned; if the read operation fails, -1 is returned.

[Return value]

Normal: Error: Actual number of bytes read -1

[Parameters]

fileno buf count Number of the file to be read Memory area to store read data Number of bytes to read



long write (long fileno, const unsigned char *buf, long count)

[Description]

- Writes data to the file indicated by the first parameter (fileno) from the memory area indicated by the second parameter (buf). The number of bytes to be written is indicated by the third parameter (count).
- If the device (disk, etc.) of the file to be written is full, only a number of bytes fewer than or equal to count bytes can be written. It is recommended that, if the number of bytes actually written is zero a certain number of times in succession, the disk should be judged to be full and an error (-1) should be returned.
- The position for file reading/writing advances by the number of bytes written. If writing is successful, the actual number of bytes written should be returned; if the write operation fails, -1 should be returned.

[Return value]

Normal:	Actual number of bytes written
Error:	-1

[Parameters]

fileno buf count Number of the file to which data is to be written Memory area containing data for writing Number of bytes to write



long lseek (long fileno, long offset, long base)

[Description]

- Sets the position within the file, in byte units, for reading from and writing to the file.
- The position within a new file should be calculated and set using the following methods, depending on the third parameter (base).
 - (1) When base is 0: Set the position at offset bytes from the file beginning
 - (2) When base is 1: Set the position at the current position plus offset bytes
 - (3) When base is 2: Set the position at the file size plus offset bytes
- When the file is a console, printer, or other interactive device, when the new offset is negative, or when in cases (1) and (2) the file size is exceeded, an error occurs.
- When the file position is set correctly, the new position for reading/writing should be returned as an offset from the file beginning; when the operation is not successful, -1 should be returned.

from the file beginning

The new position for file reading/writing, as an offset in bytes

[Return value]

Normal:

Error:

[Parameters]

filenoFile numberoffsetPosition for reading/writing, as an offset (in bytes)baseStarting-point of the offset

-1



char *sbrk (size_t size)

[Description]

- The size of the memory area to be allocated is passed as a parameter.
- When calling the sbrk routine several times, memory areas should be allocated in succession starting from lower addresses. If the memory area for allocation is insufficient, an error should occur. When allocation is successful, the address of the beginning of the allocated memory area should be returned; if unsuccessful, "(char *) -1" should be returned.

[Return value]

Normal:	
Error:	

Start address of allocated memory (char *) -1

[Parameters]

size Size of data to be allocated (1) Example of Coding Low-Level Interface Routines ****** 11 /* */ lowsrc.c: /* -----*/ /* */ RX Family Simulator/Debugger Interface Routine /* - Only standard I/O (stdin,stdout,stderr) are supported -*/ #include <string.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 _MOPENR0x1 #define _MOPENW0x2 #define _MOPENA0x4 #define _MTRUNC0x8 #define _MCREAT0x10 #define _MBIN0x20 #define _MEXCL0x40 #define _MALBUF0x40 #define _MALFIL0x80 #define _MEOF0x100 #define _MERR0x200 #define _MLBF0x400 #define _MNBF0x800 #define _MREAD0x1000 #define _MWRITE0x2000 #define _MBYTE0x4000 #define MWIDE0x8000 /* 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 */

RENESAS

```
/* 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
                                                                                   */
                                                                                    */
    /\ast 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*/
```

RENESAS

```
/* Specifies no data buffering.
                                                         * /
   stdout->_Mode | = _MNBF;
  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++ )</pre>
   ł
      /* 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
                                                         */
                              /* Open mode
                                                        */
   long 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;
}
```



```
CubeSuite+ V2.02.00
```

```
* /
/* 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 */
                           /* Written character count
    long 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. */
      }
                                                              * /
                                 /* Output to file
      else return -1;
   }
                                 /* Error
                                                              * /
   else return -1;
}
long read( long fileno, unsigned char *buf, long count )
{
     unsigned long i;
     /* Checks mode according to file number, inputs one character each, */
     /\,{}^{\star} 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		
; _cha:	rput:			
;	 t:			
_01141	MOV.L MOV.L MOV.L MOV.L MOV.L MOV.L JSR RTS	<pre>#IO_BUF,R2 R1,[R2] #1220000h,R1 #PARM,R3 R2,[R3] R3,R2 #SIM_IO,R3 R3</pre>		
;; ;cha: ;	rget:			
, _charge	t:			
	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		
	RTS	[R2],R1		
;				
; I/O]	Buffer			
PARM: IO_BUF:	.SECTION .BLKL .SECTION .BLKB	B,DATA,ALIGN=4 1 B_1,DATA 1		
	.END			

7.3.5 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</pre>
        if(_atexit_buf[i]==f)
             return 1;
    if(_atexit_count==32)
                                        // Check the limit value of number of reg-
istration
       return 1;
    else {
       atexit_buf[_atexit_count++]=f; // Register the function address
       return 0;
    }
}
```



```
Example of Preparation of a Routine for Program Termination (exit)
(2)
    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 pro-
    cedure 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 setimp function imme-
     diately before the main function was called. In order to return to the environment prior to program execution, the
     following callmain function should be created, and instead of calling the function main from the
     PowerOn_Reset_PC initial setting function, the callmain function should be called.
     A program example is shown below.
      #include <setjmp.h>
      #include <stddef.h>
     extern long _atexit_count ;
     extern void_t (*_atexit_buf[32])(void) ;
      #ifdef __cplusplus
      extern "C"
      #endif
     void _CLOSEALL(void);
     int main(void);
     extern jmp_buf _init_env ;
     int _exit_code ;
      #ifdef __cplusplus
     extern "C"
      #endif
      void exit(int code)
      {
        int i;
        _exit_code=code ;
                                              // Set the return code in _exit_code
        for(i=_atexit_count-1; i>=0; i--)// Execute in sequence the functions
           (*_atexit_buf[i])(); // registered by the atexit function
         CLOSEALL();
                                              // Close all open functions
                                              // Return to the environment saved by
        longjmp(_init_env, 1) ;
                                              // 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.

RENESAS

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
                   ;
}
```

7.4 Coding Example

This section shows an example of an actual startup program created for the simulator in the integrated development environment when the RX610 is selected as the CPU type.

(1) Source Files

The startup program consists of the files shown in table 7.4.

Table 7.4	List of Programs	Created in	Integrated D	Development	Environment

	File Name	Description
(a)	resetprg.c	Initial setting routine (reset vector function)
(b)	intprg.c	Vector function definitions
(c)	vecttbl.c	Fixed vector table *1
(d)	dbsct.c	Section initialization processing (table)
(e)	lowsrc.c	Low-level interface routine (C language part)
(f)	lowlvl.src	Low-level interface routine (assembly language part)
(g)	sbrk.c	Low-level interface routine (sbrk function)
(h)	typedefine.h	Type definition header
(i)	vect.h	Vector function header
(j)	stacksct.h	Stack size settings
(k)	lowsrc.h	Low-level interface routine (C language header)
(I)	sbrk.h	Low-level interface routine (sbrk function header)

*1) This name is called in the case of the RXv1 instruction-set architecture.

This name is changed to "Exception vector table" in the case of RXv2 instruction-set architecture.

The following shows the contents of files (a) to (l).

```
(a) resetprg.c: Initial Setting Routine (Reset Vector Function)
```

```
#include<machine.h>
#include<_h_c_lib.h>
//#include<stddef.h> // Remove the comment when you use errno
//#include <stdlib.h> // Remove the comment when you use rand()
#include"typedefine.h" // Define Types
#include"stacksct.h" // Stack Sizes (Interrupt and User)
```



```
#ifdef __cplusplus
                        // For Use Reset vector
extern "C" {
#endif
void PowerOn_Reset_PC(void);
void main(void);
#ifdef __cplusplus
}
#endif
                        // For Use SIM I/O
#ifdef __cplusplus
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 *_slptr;
                             // Remove the comment when you use strtok()
//#ifdef __cplusplus
                             // Use Hardware Setup
//extern "C" {
//#endif
//extern void HardwareSetup(void);
//#ifdef __cplusplus
//}
//#endif
//#ifdef __cplusplus
                           // Remove the comment when you use global class object
//extern "C" {
                             // Sections C$INIT and C$END will be generated
//#endif
//extern void _CALL_INIT(void);
//extern void _CALL_END(void);
//#ifdef __cplusplus
//}
//#endif
#pragma section ResetPRG
                            // output PowerOn_Reset_PC to PResetPRG section
#pragma entry PowerOn_Reset_PC
void PowerOn_Reset_PC(void)
{
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 0x0000100
                           // Let FPSW DNbit=1 (denormal as zero)
#else
#define _DENOM 0x0000000
                          // Let FPSW DNbit=0 (denormal as is)
#endif
set_fpsw(FPSW_init | _ROUND | _DENOM);
                           // Initialize Sections
_INITSCT();
```



```
// Use SIM I/O
_INIT_IOLIB();
//errno=0;
                           // Remove the comment when you use errno
//srand((_UINT)1);
                           // Remove the comment when you use rand()
//_slptr=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();
                           // Use SIM I/O
_CLOSEALL();
//_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) = {
//;0xfffffd0 Exception (Supervisor Instruction)
    Excep_SuperVisorInst,
//;0xffffffd4 Reserved
   Dummy,
//;0xfffffd8 Reserved
   Dummy,
//;0xfffffdc Exception (Undefined Instruction)
   Excep_UndefinedInst,
//;0xffffffe0 Reserved
   Dummy,
//;0xffffffe4 Exception (Floating Point)
   Excep_FloatingPoint,
//;0xfffffe8 Reserved
   Dummy,
//;0xfffffec Reserved
   Dummy,
//;0xffffff0 Reserved
   Dummy,
//;0xffffffff Reserved
   Dummy,
//;0xffffff8 NMI
   NonMaskableInterrupt,
//;0xffffffc RESET
//;<<VECTOR DATA START (POWER ON RESET)>>
//;Power On Reset PC
PowerOn_Reset_PC
//;<<VECTOR DATA END (POWER ON RESET)>>
};
```



(d) dbsct.c: Section Initialization Processing (table)

```
#include "typedefine.h"
#pragma unpack
#pragma section C C$DSEC
extern const struct {
                    /* Start address of the initialized data section in ROM */
/* End address of the initialized data section in ROM */
    _UBYTE *rom_s;
    _UBYTE *rom_e;
    _UBYTE *ram_s;
                      /* Start address of the initialized data section in RAM */
    _DTBL[] = {
}
    { __sectop("D"), __secend("D"), __sectop("R") },
    { __sectop("D_2"), __secend("D_2"), __sectop("R_2") },
    { __sectop("D_1"), __secend("D_1"), __sectop("R_1") }
};
#pragma section C C$BSEC
extern const struct {
   _UBYTE *b_s;
                          /* Start address of non-initialized data section */
                         /* End address of non-initialized data section */
   _UBYTE *b_e;
}
    \_BTBL[] = {
    { __sectop("B"), __secend("B") },
    { __sectop("B_2"), __secend("B_2") },
    { __sectop("B_1"), __secend("B_1") }
};
#pragma section
/*
** CTBL prevents excessive output of W0561100 messages when linking.
** Even if CTBL is deleted, the operation of the program does not change.
*/
_UBYTE * const _CTBL[] = {
    __sectop("C_1"), __sectop("C_2"), __sectop("C"),
    __sectop("W_1"), __sectop("W_2"), __sectop("W")
};
```

#pragma packoption



```
(e) lowsrc.c : Low-Level Interface Routine (C Language Part)
#include <string.h>
#include <stdio.h>
#include <stddef.h>
#include "lowsrc.h"
#define STDIN 0
#define STDOUT 1
#define STDERR 2
#define FLMIN 0
#define _MOPENR0x1
#define _MOPENW0x2
#define _MOPENA0x4
#define _MTRUNC0x8
#define _MCREAT0x10
#define _MBIN0x20
#define _MEXCL0x40
#define _MALBUF0x40
#define _MALFIL0x80
#define _MEOF0x100
#define _MERR0x200
#define _MLBF0x400
#define _MNBF0x800
#define _MREAD0x1000
#define _MWRITE0x2000
#define _MBYTE0x4000
#define _MWIDE0x8000
#define O_RDONLY0x0001
#define O_WRONLY0x0002
#define O_RDWR0x0004
#define O_CREAT0x0008
#define O_TRUNC0x0010
#define O_APPEND0x0020
#define CR 0x0d
#define LF 0x0a
extern const long _nfiles;
char flmod[IOSTREAM];
unsigned char sml_buf[IOSTREAM];
#define FPATH_STDIN
                         "C:\\stdin"
#define FPATH_STDOUT
                         "C:\\stdout"
#define FPATH_STDERR
                         "C:\\stderr"
extern void charput(unsigned char);
extern unsigned char charget(void);
#include <stdio.h>
FILE *_Files[IOSTREAM];
char *env_list[] = {
    "ENV1=temp01",
    "ENV2=temp02",
    "ENV9=end",
    '\0'
};
char **environ = env_list;
```

void _INIT_IOLIB(void)



```
{
    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++ )</pre>
    ł
        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)
{
```



```
i;
    long
    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)

```
_charput
     .GLB
        _charget
     .GLB
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

ZOD D2
     JSR
          R3
     RTS
; _charget:
_charget:
    MOV.L
           #1210000h,R1
    MOV.L
MOV.L
          #IO_BUF,R2
          #PARM,R3
          R2,[R3]
    MOV.L
          R3,R2
    MOV.L
    MOV.L
           #SIM_IO,R3
    JSR
           R3
    MOV.L
           #IO_BUF,R2
           [R2],R1
    MOVU.B
    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)
                                                             */
                                                            */
/*
                                      (Failure)
                -1
_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
                                                           */
                                       /* End address update
                                                           */
      brk += size;
   }
   return p;
}
```

(h) typedefine.h: Type Definition Header

```
typedef signed char _SBYTE;
typedef unsigned char _UBYTE;
typedef signed short _SWORD;
typedef unsigned short _UWORD;
typedef signed int _SINT;
typedef unsigned int _UINT;
typedef signed long _SDWORD;
typedef unsigned long _UDWORD;
typedef signed long long _SQWORD;
typedef unsigned long long _UQWORD;
```


(i) vect.h: Vector Function Header

```
// Exception (Supervisor Instruction)
#pragma interrupt (Excep_SuperVisorInst)
void Excep_SuperVisorInst(void);
```

// Exception (Undefined Instruction)
#pragma interrupt (Excep_UndefinedInst)
void Excep_UndefinedInst(void);

// Exception (Floating Point)
#pragma interrupt (Excep_FloatingPoint)
void Excep_FloatingPoint(void);

```
// NMI
#pragma interrupt (NonMaskableInterrupt)
void NonMaskableInterrupt(void);
```

// Dummy
#pragma interrupt (Dummy)
void Dummy(void);

```
// BRK
#pragma interrupt (Excep_BRK(vect=0))
void Excep_BRK(void);
```

//;<<VECTOR DATA START (POWER ON RESET)>>
//;Power On Reset PC
extern void PowerOn_Reset_PC(void);
//;<<VECTOR DATA END (POWER ON RESET)>>

(j) stacksct.h: Stack Size Settings

// <code>#pragma stacksize su=0x100</code> // Remove the comment when you use user stack <code>#pragma stacksize si=0x300</code>

(k) lowsrc.h: Low-Level Interface Routine (C Language Header)

```
/*Number of I/O Streams*/
#define IOSTREAM 20
```

(I) sbrk.h: Low-Level Interface Routine (sbrk Function Header)

/* Size of area managed by sbrk */
#define HEAPSIZE 0x400

(m) Execution Commands

The following shows an example of commands for building these files. In this example, the name of the user program file (containing the **main** function) is UserProgram.c, and the body of the file names (names excluding extensions) for the load module or library to be created is LoadModule.



lbgrx -isa=rxv1 -output=LoadModule.lib ccrx -isa=rxv1 -output=obj UserProgram.c ccrx -isa=rxv1 -output=obj intprg.c ccrx -isa=rxv1 -output=obj vecttbl.c ccrx -isa=rxv1 -output=obj dbsct.c ccrx -isa=rxv1 -output=obj lowsrc.c asrx -isa=rxv1 -output=obj lowsrc.c asrx -isa=rxv1 lowlv1.src ccrx -isa=rxv1 -output=obj sbrk.c rlink -rom=D=R,D_1=R_1,D_2=R_2 -list=LoadModule.map start=B_1,R_1,B_2,R_2,B,R,SI/01000,PResetPRG/ 0FFFF8000,C_1,C_2,C,C\$*,D_1,D_2,D,P,PIntPRG,W*,L/0FFFF8100,FIXEDVECT/0FFFFFFD0 library=LoadModule.lib -output=LoadModule.abs UserProgram.obj resetprg.obj intprg.obj vecttbl.obj dbsct.obj lowsrc.obj lowlv1.obj sbrk.obj rlink -output=LoadModule.sty -form=stype -output=LoadModule.mot LoadModule.abs

7.5 Usage of PIC/PID Function

This section gives an overview of the PIC/PID function and describes how to create startup programs when using the PIC/PID function.

The PIC/PID function enables the code and data in the ROM to be reallocated to desired addresses without re-linkage even when the allocation addresses have been determined through previously completed linkage.

PIC stands for position independent code, and PID stands for position independent data. The PIC function generates PIC and the PID function generates PID; here, these functions are collectively called the PIC/PID function.

7.5.1 Terms Used in this Section

(1) Master and Application

In the PIC/PID function, a program whose code or data in the ROM has been converted into PIC or PID is called an application, and the program necessary to execute an application is called the master. The master executes the application initiation processing, and also provides the shared libraries called from applications and RAM areas for applications. PIC and PID are included only in applications; the master does not have them.

(2) Shared Library

A group of functions in the master, which can be called from multiple applications.

(3) Jump Table A program through which applications can call shared libraries.

7.5.2 Function of Each Option

The following describes the options related to the PIC/PID function. For details of each option function, refer to the respective option description of RX Build.

(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.

7.5.3 Restrictions on Applications

- (1) RAM Areas The PID function cannot be applied to the RAM area.
- (2) Simultaneous Execution of Applications When the PIC/PID function is used, multiple copies of a single application can be stored in the ROM and each copy can be executed. However, copies of a single application cannot be executed at the same time because the RAM areas for them overlap each other.
- (3) Startup

The standard startup program (created by the integrated development environment as described in section 7.3 Startup Program Creation) cannot be used to start up an application without change. Create a startup program as described in 7.5.7 Application Startup.

7.5.4 System Dependent Processing Necessary for PIC/PID Function

The following processing should be prepared by the user depending on the system specifications.

- 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.
- 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.

7.5.5 Combinations of Code Generating Options

When the master and application are built, the option settings related to the PIC/PID function should be matched between the objects that compose the master and application.

The following shows the rules for specifying options for each object compilation and the conditions of option settings in other objects that can be linked.

(1) Master

When building the master, specify the PIC/PID function options as shown in table 7.5.

Table 7.5	Rules for Specifying PIC/PID Function Options in Master	
		_

Option Name	For Compilation	Conditions on Setting the Option for Linkable Objects
pic	× Not allowed	pic is not specified
pid	× Not allowed	pid is not specified



nouse_pid_register	O Can be specified except the standard library and set- ting PID register of the startup program	No conditions
fint_register	O Can be specified	<pre>fint_register with the same parameters must be specified</pre>
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 7.6.

Table 7.6	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	\times 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 speci- fied

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 7.7.

Table 7.7	Rules for Combinations of PIC/PID	Function Options between Master	and Application
-----------	-----------------------------------	---------------------------------	-----------------

Options in Application	Options in Master
pic	No conditions
pid nouse_pid_register is necessary if application calls fun of master	
fint_register fint_register with the same parameters is necessary	
base	base* with the same parameters is necessary

Note * When **pid** is specified, **base=rom=<register>** is not allowed.

7.5.6 Master Startup

The processing necessary to start up the master is the same as that for a usual program that does not use the PIC/PID function except for the two processes described below. Add these two processes to the startup processing created according to section 7.3, Startup Program Creation.

- (1) Initiation of and Return from Application Set up the PID register in the main function and branch to the PIC entry address to initiate the application. In addition, a means for returning from the application to the master should be provided.
- Reference to Shared Library Functions to be Used
 The shared libraries to be used by the application should be referred to also by the master in advance.
 The following shows an example for calling a PIC/PID application from the main function.
 This example assumes the following conditions:
 - After application execution, control can be returned to the master through the RTS instruction.



- The application does not pass a return value.
- The PID initiation address (PIC_entry) and PID start address (PID_address) for the application are known and fixed when the master is built.
- R13 is used as the PID register.
- Initialization of the section areas on the application side is not done on the master side.

- The application uses only the **printf** function as the shared library. Example:

```
/* Master-Side Program */
/* Initiates the PIC/PID application. */
/* (For the system that the application does not pass */
/* a return value and execution returns through RTS) */
#include <stdio.h>
#pragma inline asm Launch PICPID Application
void Launch PICPID Application(void *pic entry, void *pid address)
{
   MOV.L
            R2,___PID_R13
   JSR
            R1
}
int main()
{
                    = (void*)0x500000; /* PIC initiation address */
   void *PIC_entry
   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 */
```

7.5.7 Application Startup

Specify the following in the application. The items marked with **[Optional]** may be unnecessary in some cases.

- Preparation of Entry Point (PIC Initiation Address)This is the address from which the application is initiated.
- Initialization of Stack Pointer [Optional] This processing is not necessary when the application shares the stack with the master. When necessary, add appropriate settings by referring to section 7.3.2 (2).
- Initialization of General Registers Used as Base Registers [Optional] This processing is not necessary when no base register is used. When necessary, add appropriate settings by referring to section 7.3.2 (3).
- Initialization Processing of Sections [Optional]
 This processing is not necessary when the master initializes them.
 When necessary, add appropriate settings by referring to the example shown later.
 Note that the processing described in section 7.3.2 (5) cannot be used without change.
- (5) Initialization Processing of Libraries [Optional] This processing is not necessary when no standard library is used. When necessary, add appropriate settings by referring to section 7.3.2 (6). Initialization of PSW for main Function Execution [Optional] Specify interrupt masks or move to the user mode as necessary. Add appropriate settings by referring to sections 7.3.2 (8) and 7.3.2 (9).
- (6) User Program Execution



```
Execute the main function.
Specify the processing by referring to section 7.3.2 (10).
```

The following shows an example of application startup. The processing is divided into three files.

- startup_picpid.c: Body of the startup processing.

```
    initsct_pid.src: Section initialization for PID; _INITSCT_PID.
    This is created by modifying the _INITSCT function described in section 7.3.2 (5) to support the PID function.
    "_PID_REG" in the program will be converted into PID register when the assembling.
```

```
    initiolib.c; Contains _INITLIB, which initializes the standard libraries.
    This is created by modifying the code described in section 7.3.2 (6) to be used for the application.
    [startup_picpid.c]
```

```
// Initialization Processing Described in Section 7.3.2(5)
#pragma section C C$DSEC //Section name is set to C$DSEC
const struct {
   void *rom_s; //Start address member of the initialized data section in ROM
   void *rom_e; //End address member of the initialized data section in ROM
   void *ram_s; //Start address member of the initialized data section in RAM
} DTBL[] = {__sectop("D"), __secend("D"), __sectop("R")};
#pragma section C C$BSEC //Section name is set to C$BSEC
const struct {
   void *b_s; //Start address member of the uninitialized data section
    void *b_e; //End address member of the uninitialized data section
} BTBL[] = {__sectop("B"), __secend("B")};
extern void main(void);
extern void _INITLIB(void); // Library initialization processing described
                            //in section 7.3.2 (6)
#pragma entry application_pic_entry
void application_pic_entry(void)
{
    _INITSCT_PICPID();
    INITLIB();
    main();
}
```



[initsct_pid.src]

```
; Section Initialization Routine for PID Support
; ** Note ** Check the PID register.
; This code assumes that R13 is used as the PID register. If another
; register is used as the PID register, modify the description related to R13
; in the following code to the register assigned as the PID register
; in your system.
    .glb __INITSCT_PICPID
.glb __PID_TOP
    .section C$BSEC,ROMDATA,ALIGN=4
    .section C$DSEC,ROMDATA,ALIGN=4
    .section P,CODE
__INITSCT_PICPID:
                                ; function: _INITSCT
   .STACK
            __INITSCT_PICPID=28
             R1-R6
   PUSHM
             #-__PID_TOP,__PID_REG,R6 ; How long distance PID moves
   ADD
;;;
;;; clear BBS(B)
;;;
              #TOPOF C$BSEC, R6, R4
   ADD
             #SIZEOF C$BSEC, R4, R5
   ADD
   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)
;;;
              #TOPOF C$DSEC, R6, R4
   ADD
              #SIZEOF C$DSEC, R4, R5
   ADD
   BRA
              next_loop3
loop3:
   MOV.L
             [R4+], R2
   MOV.L
             [R4+], R3
   MOV.L
             [R4+], R1
   CMP
             R2, R3
   BLEU
             next_loop3
             R2, R3
   SUB
                          ; Adjust for real address of PID
   ADD
              R6, R2
   SMOVF
next_loop3:
              R4, R5
   CMP
    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
}
```



8. Referencing Compiler and Assembler

For confirming how to handle arguments when a program is called by the CCRX, please see following sections. This chapter describes some examples of C/C++ program and Assembly program calls each other.

3.4 Function Calling Interface

3.4.1 Rules Concerning the Stack

3.4.2 Rules Concerning Registers

3.4.3 Rules Concerning Setting and Referencing Parameters

3.4.4 Rules Concerning Setting and Referencing Return Values

3.4.5 Examples of Parameter Allocation

8.1 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)
- Method for referencing assembly program external names in C/C++ programs (1) 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) C/C++ program (reference) extern int a,b; .glb _a, _b .SECTION D, ROMDATA, ALIGN=4 a: .LWORD 1 void f() _b: .LWORD 1 { .END 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)	Assembly program (reference)
int a;	.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

 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



9. Usage Notes

This chapter describes the points to be noted when using the CCRX.

9.1 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)
\{\ldots\}
      <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)
\{\ldots\}
```

(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.



```
CubeSuite+ V2.02.00
```

```
}
```

(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]
sub(++i, i)

;The value on the left side differs depending on whether the right side of the assignment expression is evaluated first. ;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

result = old_style(1,2,3,4,5);

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*/
:
                              /* area indicated by the parameter may change*/
strcat(p, "abc");
  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*/
:
                              /* is not an error*/
i=10;
 (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 $a\cos(1.0 - \epsilon)$

double precision 2^{-39} ($\epsilon = 2^{-33}$) single precision 2^{-21} ($\epsilon = 2^{-19}$)

	Absolute error fo	r asin(1.0 – ε)	double precision 2^{-39} ($\varepsilon = 2^{-28}$) single precision 2^{-21} ($\varepsilon = 2^{-16}$)
(7)	Codes that May I A code continuou may be deleted a order for accesse	be Deleted by Optim usly referencing the as redundant codes as to always be guar	ization same variable or a code containing an expression whose result is not used at optimization by the compiler. Variables should be declared with volatile in ranteed.
	<example></example>		
[1] b	=a;	/* The expres /* as redunda	sion in the first line may be deleted*/ nt code*/
b	=a;		
[2] w	hile(1)a;	/* The refere /* statement	nce to variable a and the loop */ may be deleted as redundant code*/
(8)	Differences betw In the C99, selec to differ in the C8	een C89 Operation ation statements and 9 and C99.	and C99 Operation repeat statements are enclosed in curly brackets { }. This causes operations
enum int g { i	<pre><example> {a,b}; (void) f(!sizeof(enumeraturn a; eturn b;</example></pre>	n{b,a}))	
}			
If th enum int g {	<pre>e above code : {a,b}; (void) { if(!sizeo: return a; } return b;</pre>	is compiled wit f(enum{b,a}))	h -lang=c99 specified, it is interpreted as follows:
}			
	g()=0 in -lang=c	becomes g()=1 in -I	ang=c99.
(9)	Operations and T The result of any values must not of affected by other In the standard C according to the The following exa	Type Conversions The operation or type conserved by the conserved of the conserved by the conserved by the conditions such as conditions such as conditions such as conserved by the conserved by	hat Lead to Overflows proversion must be within the allowed range of values for the given type (i.e. flow does occur, the result of the operation or type conversion may be compiler options. It of an operation that leads to an overflow is undefined and thus may differ f compilation. Ensure that no operations in a program will lead to an overflow. problem.
float	Example: Type c f = 21474836	onversion from float	to unsigned short

The value of **ui2**, which is acquired as the result of executing **ex1func**, depends on whether **-fpu** or **-nofpu** has been specified.

```
-fpu (with the FPU): ui2 = 65535
-nofpu (without the FPU): ui2 = 0
```

This is because the method of type conversion from **float** to **unsigned short** differs according to whether **–fpu** or **–nofpu** has been specified.

(10) Symbols That Contain Two or More Underscores (__)

Symbols must not contain sequences of two or more underscores. Even though the code generated in such cases seems normal, the symbol names may be mistaken as different C++ function names when they are output as link-age-map information.

```
Example:
int sample__Fc(void) { return 0; }
```

This will be output to the linkage map as sample(char) rather than _sample_Fc.

9.2 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 cvalue 1=0;
   // Gives initial value
extern const cvalue 2 = 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
}
```

9.3 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.
 - The five options isa, cpu, endian, base, and fint_register should be specified in the same way in the compiler, assembler, and library generator.
 - 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.

9.4 Compatibility with an Older Version or Older Revision

The effect of the compatibility regarding a version change or revision change is described here.

9.4.1 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 9.1.



No.	Item	Specification	Function	Changed Contents	
				Item	Details
1	User stack	void set_usp(void *data)	USP setting	Parameter	unsigned long \rightarrow void *
2	(USP)	void *get_usp(void)	USP reference	Return value	unsigned long \rightarrow void *
3	Interrupt	void set_isp(void *data)	ISP setting	Parameter	unsigned long \rightarrow void *
4	(ISP)	void *get_isp(void)	ISP reference	Return value	unsigned long \rightarrow void *
5	Interrupt	void set_intb (void *data)	INTB setting	Parameter	unsigned long \rightarrow void *
6	ter (INTB)	void *get_intb(void)	INTB reference	Return value	unsigned long \rightarrow void *
7	Backup PC	<pre>void set_bpc(void *data)</pre>	BPC setting	Parameter	unsigned long \rightarrow void *
8	(BPC)	<pre>void *get_bpc(void)</pre>	BPC reference	Return value	unsigned long \rightarrow void *
9	Fast inter-	void set_fintv(void *data)	FINTV setting	Parameter	unsigned long \rightarrow void *
10	rupt vector register (FINTV)	void *get_fintv(void)	FINTV reference	Return value	unsigned long \rightarrow void *

Table 9.1	List of Intrinsic	Functions Whose	Type is	Changed
-----------	-------------------	------------------------	---------	---------

Due to this change, a program using the above functions in V.1.00 may generate a warning or an error about invalid types. In this case, add or delete the cast to correct the types.

An example of a startup program normally used in V.1.00 is shown below. This example will output warning message W0520167 in V.1.01, but this warning can be avoided by deleting the cast to correct the type.

<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$VECT")); //Warning W0520167 is output
         . . .
}
      [Example of code changed to match V.1.01]
#include <machine.h>
#pragma entry Reset_Program
void PowerON_Reset_PC(void)
{
         . . .
        set_intb(__sectop("C$VECT")); //Cast (unsigned long) is deleted
         . . .
}
```

(2) Adding Section L (section Option and Start Option)

V.1.01 is provided with section **L** which is used for storing literal areas, such as, string literal. Since the number of sections has increased and section **L** is located at the end at linkage, the optimizing linkage editor may output address error F0563100 in some cases. To avoid such an error, adopt either one of the following methods.

- (a) Add L to the section sequence specified with the Start option of the optimizing linkage editor at linkage.
 <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/0FFFFFD0
```

[Changed example (L is added after C)]



```
-start=B_1,R_1,B_2,R_2,B,R,SU,SI/01000,PResetPRG/0FFF8000,C_1,C_2,
C,L,C$*,D*,P,PIntPRG,W*/0FFFF8100,FIXEDVECT/0FFFFFD0
```

(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.

9.4.2 Compatibility with V.1.01 and V.1.02

(1) Restriction That Applies to Operation of the Linkage Editor When the -merge_files Option of the Compiler Has been Used

When an object module file created by the compiler with the **-merge_files** option specified is to be linked, correct operation is not guaranteed if the **-delete**, **-rename**, or **-replace** option is specified.

(2) Note on Generation of Code That Corresponds to if Statements When optimize=0 In this version of this compiler, if statements where the conditional expression has a constant value and statements that will accordingly never be executed are not reflected in the output code whether or not optimize=0 is specified.

In the examples below, lines marked [Deleted] are not reflected at the time of code generation.

Example 1: Expression that produces a constant value

```
int a,b,c;
void func01(void)
{
    if (1+2) { /* [Deleted] */
        /* Executed */
        a = b;
    } else {
        /* Never executed */
        a++; /* [Deleted] */
        b = c; /* [Deleted] */
    }
}
```

Example 2: Constant expressions that include symbolic addresses are also treated as constant expressions. void fl(void), f2(void);

```
void func02(void)
{
    if (f1==0) { /* [Deleted] */
        /* Never executed */
        f2(); /* [Deleted] */
    } else {
        /* Executed */
        f1();
    }
}
```

(3) Differences in Assembly Source Code Output by -show=source There are the following differences in the assembly source code to be output by this version of this compiler when -show=source is specified.

- .LINE is not displayed unless -debug has been specified.
- **#include** statements are not expanded.
- The instruction that corresponds to source code that follows #line may be incorrect.



A. WINDOW REFERENCE

This section describes the window, panel, and dialog boxes related to coding.

A.1 Description

Below is a list of the window, panel, and dialog boxes related to coding.

Table A.1	List of Window/Panel/Dialog Boxes
-----------	-----------------------------------

Window/Panel/Dialog Box Name	Function Description
Editor panel	This panel is used to display and edit files.
Encoding dialog box	This dialog box is used to select a file-encoding.
Go to Line dialog box	This dialog box is used to move the caret to a specified source line.
Print Preview window	This window is used to preview the source file before printing.
Open File dialog box	This dialog box is used to open a file.



Editor panel

This panel is used to display and edit files.

When opened the file encoding and newline code is automatically detected and retained when it is saved. You can open a file with a specific encoding selected in the Encoding dialog box. If the encoding and newline code is specified in the Save Settings dialog box then the file is saved with those settings.

This panel can be opened multiple times (max. 100 panels).

- Remarks 1. When a project is closed, all of the Editor panels displaying a file being registered in the project are closed.
- Remarks 2. When a file is excluded from a project, the Editor panel displaying the file is closed.
- Remarks 3. A message is shown when the downloaded load module file is older than the source file to be opened. This is due to the debug information not matching the source code being viewed.

Figure A.1 Editor Panel



The following items are explained here.

- [How to open]
- [Description of each area]
- [Toolbar]
- [[File] menu (Editor panel-dedicated items)]
- [[Edit] menu (Editor panel-dedicated items)]
- [[Window] menu (Editor panel-dedicated items)]
- [Context menu]



[How to open]

- On the Project Tree panel, double click a file.
- On the Project Tree panel, select a source file, and then select [Open] from the context menu.
- On the Project Tree panel, select [Add] >> [Add New File...] from the context menu, and then create a text file or source file.

[Description of each area]

(1) Title bar

The name of the open text file or source file is displayed. Marks displayed at the end of the file name indicate the following:

Mark	Description
*	The text file has been modified since being opened.
(Read only)	The opened text file is read only.

(2) Column header

The title of each column on the Editor panel is displayed (hovering the mouse cursor over this area displays the title name).

Display	Title Name	Description
Line	Line	Displays line numbers (see "(4) Line number area").
(No dis- play)	Selection	The display is colored to reflect the state in terms of saving of the state of editing (see "(4) Line number area").

Remark Show/hide of the column header can be switched by the setting of the toolbar.

(3) Splitter bars

You can split the Editor panel by using the horizontal and vertical splitter bars within the view. This panel can be split up to two times vertically, and two times horizontally.

- To split this panel, drag the splitter bar down or to the right to the desired position, or double-click any part of the splitter bar.
- To remove the split, double-click any part of the splitter bar.



🝯 main.c							
88	void main	(void)	~	105	void	func1()	~
89 6	∃{		_	106	⊡ {		_
90 8		* Start user	: e -	107		UINT i;	
91				108			
92	TI	<pre>MP0_Start();</pre>		109		for (i =	0; i
93	TI	<pre>MP1_Start();</pre>		110		f	unc1a 💻
94	A	D_Start ();		111		}	
95				112	}		
96	W	hile (1U)	~	113			~
<			>	<	1111		>
122	void f	unc2 ()	^	140		a = &globa	al ch 🔼
123 🗄	∃{			141			
124	U.	INT i;		142	Ē	/* Comment	t */
125				143	Ē	/* Comment	t */
126	f	or (i = 0; i		144	Ē	/* Comment	t */
127		func2	a	145			
128	}		_	146		global_a+	+; 👝
129	}			147		global_b+	+;
130			~	148		global_c+	+; 🗸
<			>	<			>

Figure A.2 Editor Panel (Vertical/Horizontal Two-way Split View)

(4) Line number area

This area displays the line number of the opened text file or source file. On each line there is an indicator that shows the line modification status.

<1> 88	void main(void)
<2> 90	∃{ /* Start user code. Do not
91	-
92	<pre>TMP0_Start();</pre>
93	<pre>TMP1_Start();</pre>
94	AD Start ();
95	_
96	while (1U)

<1>	This means new or modified line but unsaved.
<2>	This means new or modified line and saved. To erase this mark, close the panel, and then open this source file again.

(5) Characters area

This area displays character strings of text files and source files and you can edit it. This area has the following functions.

(a) Code outlining

This allows you to expand and collapse source code blocks so that you can concentrate on the areas of code which you are currently modifying or debugging. This is only available for only C and C++ source file types. This is achieved by clicking the plus and minus symbols to the left of the Characters area. Types of source code blocks that can be expanded or collapsed are:

Open and close braces ('{' and '}')	•
Multi-line comments ('/*' and '*/')	* (/**/)

Pre-processor statements ('if', 'elif', 'else', 'endif')	∃ #if[Preprocessor block]
	■#elif[Preprocessor block]
	⊞#else[Preprocessor block]
	#endif

- (b) Character editing Characters can be entered from the keyboard. Various shortcut keys can be used to enhance the edit function.
- (c) Tag jump

If the information of a file name, a line number and a column number exists in the line at the caret position, selecting [Tag Jump] from the context menu opens the file in the Editor panel and jumps to the corresponding line and the corresponding column (if the target file is already opened in the Editor panel, you can jump to the panel).

(d) File monitor

The following function for monitoring is provided to manage source files.

- If the contents of the currently displayed file is changed (including renaming or deleteing) without using Cube-Suite+, a message will appear asking you whether you wish to update the file or not.

- If the contents of the currently displayed file have been changed without using CubeSuite+, a message will appear asking you whether you wish to save the file or not.
- (e) Selecting blocks
 - You can select a block that consists of multiple lines by any one of the following methods:
 - Drag the left-mouse button while holding down the [Alt] key
 - Press the [Right], [Left], [Up], or [Down] key while holding down the [Alt] + [Shift] key



Remark

K Editing of the selected block can be done by using [Cut], [Copy], [Paste], or [Delete] in the [Edit] menu.

(f) Zoom in or out on a view

You can zoom in and out of the editor view by using the [Ctrl] key + mouse-wheel combination.

- Using the [Ctrl] key + mouse-wheel forward will zoom into the view, making the contents larger and easier to see (max. 300%).
- Using the [Ctrl] key + mouse-wheel backward will zoom out of the view, making the contents smaller (min. 50%).

Remark The following items can be customized by setting the Option dialog box.

- Display fonts

- Tab interval
- Show or hide whitespace marks
- Colors of reserved words and comments



[Toolbar]

-		
WP-LLP	50	Toggles between normal (default) and mixed display mode, as the display mode of this panel. Note that this item is enabled only when connected to the debug tool and the downloaded source file is opened in this panel.
(2)		Toggles between source (default) and instruction level, as the unit in which the program is step- executed. Note that this item is enabled only when connected to the debug tool and the mixed display mode is selected.
→		Displays the current PC position. Note that this item is enabled only when connected to the debug tool.
C.		Forwards to the position before operating [Context menu] >> [Back To Last Cursor Position]. Note that this item is disabled when connected to the debug tool and the mixed display mode is selected.
	0	Goes back to the position before operating [Context menu] >> [Jump to Function]. Note that this item is disabled when connected to the debug tool and the mixed display mode is selected.
С	Columns	The following items are displayed to show or hide the columns or marks on this panel. Remove the check to hide the items (all the items are checked by default).
	Line Number	Shows the line number, in the line number area.
	Selection	Shows the mark that indicates the line modification status, in the line number area.

[[File] menu (Editor panel-dedicated items)]

The following items are exclusive for the [File] menu in the Editor panel (other items are common to all the panels).

Close file name	Closes the currently editing the Editor panel. When the contents of the panel have not been saved, a confirmation message is shown.
Save file name	Overwrites the contents of the currently editing the Editor panel. Note that when the file has never been saved or the file is read only, the same oper- ation is applied as the selection in [Save <i>file name</i> As].
Save file name As	Opens the Save As dialog box to newly save the contents of the currently editing the Editor panel.
file name Save Settings	Opens the Save Settings dialog box to change the encoding and newline code of the current focused source file in the currently editing Editor panel.
Print	Opens the Print dialog box of Windows for printing the contents of the currently edit- ing the Editor panel.
Print Preview	Opens the Print Preview window to preview the file contents to be printed.

[[Edit] menu (Editor panel-dedicated items)]

The following items are exclusive for the [Edit] menu in the Editor panel (all other items are disabled).

Undo	Cancels the previous operation on the Editor panel and restores the characters and the caret position (max 100 times).
Redo	Cancels the previous [Undo] operation on the Editor panel and restores the charac- ters and the caret position.
Cut	Cuts the selected characters and copies them to the clip board. If there is no selection, the entire line is cut.



-		
Сору		Copies the contents of the selected range to the clipboard as character string(s). If there is no selection, the entire line is copied.
Paste		Inserts (insert mode) or overwrites (overwrite mode) the characters that are copied on the clip board into the caret position. When the contents of the clipboard are not recognized as characters, the operation is invalid. The mode selected for the current source file is displayed on the status bar.
C	Delete	Deletes one character at the caret position. When there is a selection area, all the characters in the area are deleted.
S	Select All	Selects all the characters from the beginning to the end in the currently editing text file.
F	ind	Opens the Find and Replace dialog box with selecting [Quick Find] tab.
Replace		Opens the Find and Replace dialog box with selecting [Quick Replace] tab.
Go To		Opens the Go to Line dialog box to move the caret to the specified line.
Outlining		Displays a cascading menu for controlling expand and collapse states of source file outlining (see "(a) Code outlining").
	Collapse to Definitions	Collapses all nodes that are marked as implementation blocks (e.g. function definitions).
	Toggle Outlining Expansion	Toggles the current state of the innermost outlining section in which the cursor lies when you are in a nested collapsed section.
	Toggle All Outlining	Toggles the collapsed state of all outlining nodes, setting them all to the same expanded or collapsed state. If there is a mixture of collapsed and expanded nodes, all nodes will be expanded.
	Stop Outlining	Stops code outlining and remove all outlining information from source files.
	Start Automatic Outlining	Starts automatic code outlining and automatically displayed in supported source files.
Advanced		Displays a cascading menu for performing an advanced operation for the Editor panel.



Increase Line Indent	Increases the indentation of the current cursor line by one tab.
Decrease Line Indent	Decreases the indentation of the current cursor line by one tab.
Uncomment Lines	Removes the first set of line-comment delimiters from the start of the current cursor line, appropriate to the current language (e.g. C++). This operation will only be available when the language of the current source file has line-comment delimiters specified (e.g. C++).
Comment Lines	Places line-comment delimiters at the start of the current cursor line, appropriate to the current language (e.g. C++). This operation will only be available when the language of the current source file has line-comment delimiters specified (e.g. C++).
Convert Tabs to Spaces	Converts all tabs on the current cursor line into spaces.
Convert Spaces to Tabs	Converts each set of consecutive space characters on the current line to tab charac- ters, but only for those sets of spaces that are at least equal to one tab size.
Tabify Selected Lines	Tabifies the current line, causing all spaces at the start of the line (prior to any text) to be converted to tabs where possible.
Untabify Selected Lines	Untabifies the current line, causing all tabs at the start of the line (prior to any text) to be converted to spaces.
Make Uppercase	Converts all letters within the selection to uppercase.
Make Lowercase	Converts all letters within the selection to lowercase.
Toggle Character Casing	Toggles the character cases (uppercase / lowercase) of all letters within the selec- tion.
Capitalize	Capitalizes the first character of every word within the selection.
Delete Horizontal Whitespace	Deletes any excess white space either side of the cursor position, leaving only one whitespace character remaining. If there the cursor is within a word or not surrounded by whitespace, this operation will have no effect.
Trim Trailing Whitespace	Deletes any trailing whitespace that appears after the last non-whitespace character on the cursor line.
Delete Line	Completely delete the current cursor line.
Duplicate Line	Duplicates the cursor line, inserting a copy of the line immediately after the cursor line.
Delete Blank Lines	Deletes the line at the cursor if it is empty or contains only whitespace.
1	

[[Window] menu (Editor panel-dedicated items)]

The following items are exclusive for the [Window] menu in the Editor panel (other items are common to all the panels).

Split	Splits the active Editor panel horizontally. Only the active Editor panel can be split. Other panels will not be split. A panel can be split up to two times.
Remove Split	Removes the split view of the Editor panel.

[Context menu]

[Title bar area]

Close Panel	Close the currently selected panel.
Close All but This	Except for the currently selected panel, close all other panels being displayed in the same panel display area as the selected panel.



Save file name	Saves the contents of the file.
Copy Full Path	Copies the absolute path of the file to the clipboard.
Open Containing Folder	Opens the folder where the text file is saved in Explorer.
New Horizontal Tab Group	The area for the display of active panels is evenly divided into two areas in the hori- zontal direction, and the panels are displayed as a new group of tabbed pages. Only one panel is active in the new group. The area may be divided into up to four panels. This item is not displayed in the following cases.
	- Only one panel is open.
	- The group has already been divided in the vertical direction.
	- The group has already been divided into four panels.
New Vertical Tab Group	The area for the display of active panels is evenly divided into two areas in the verti- cal direction, and the panels are displayed as a new group of tabbed pages. Only one panel is active in the new group. The area may be divided into up to four panels. This item is not displayed in the following cases.
	- Only one panel is open.
	- The group has already been divided in the horizontal direction.
	- The group has already been divided into four panels.
Go to Next Tab Group	When the display area is divided in the horizontal direction, this moves the displayed panel to the group under that displaying the selected panel. When the display area is divided in the vertical direction, this moves the displayed panel to the group to the right of that displaying the selected panel. This item is not displayed if there is no group in the given direction.
Go to Previous Tab Group	When the display area is divided in the horizontal direction, this moves the displayed panel to the group over that displaying the selected panel. When the display area is divided in the vertical direction, this moves the displayed panel to the group to the left of that displaying the selected panel. This item is not displayed if there is no group in the given direction.

[Characters area]

Cut	Cuts the selected character string and copies it to the clipboard. If there is no selection, the entire line is cut.
Сору	Copies the contents of the selected range to the clipboard as character string(s). If there is no selection, the entire line is copied.
Paste	Inserts (insert mode) or overwrites (overwrite mode) the characters that are copied on the clip board into the caret position. When the contents of the clipboard are not recognized as characters, the operation is invalid. The mode selected for the current source file is displayed on the status bar.
Find	Opens the Find and Replace dialog box with selecting [Quick Find] tab.
Go To	Opens the Go to Line dialog box to move the caret to the specified line.
Jump to Function	Jumps to the function that is selected or at the caret position regarding the selected characters and the words at the caret position as functions.
Tag Jump	Jumps to the corresponding line and column in the corresponding file if the informa- tion of a file name, a line number and a column number exists in the line at the caret position (see "(c) Tag jump").
Advanced	Displays a cascading menu for performing an advanced operation for the Editor panel.



Increases the indentation of the current cursor line by one tab.
Decreases the indentation of the current cursor line by one tab.
Removes the first set of line-comment delimiters from the start of the current cursor line, appropriate to the current language (e.g. C++). This operation will only be available when the language of the current source file has line-comment delimiters specified (e.g. C++).
Places line-comment delimiters at the start of the current cursor line, appropriate to the current language (e.g. C++). This operation will only be available when the language of the current source file has line-comment delimiters specified (e.g. C++).
Converts all tabs on the current cursor line into spaces.
Converts each set of consecutive space characters on the current line to tab charac- ters, but only for those sets of spaces that are at least equal to one tab size.
Tabifies the current line, causing all spaces at the start of the line (prior to any text) to be converted to tabs where possible.
Untabifies the current line, causing all tabs at the start of the line (prior to any text) to be converted to spaces.
Converts all letters within the selection to uppercase.
Converts all letters within the selection to lowercase.
Toggles the character cases (uppercase / lowercase) of all letters within the selec- tion.
Capitalizes the first character of every word within the selection.
Deletes any excess white space either side of the cursor position, leaving only one whitespace character remaining. If there the cursor is within a word or not surrounded by whitespace, this operation will have no effect.
Deletes any trailing whitespace that appears after the last non-whitespace character on the cursor line.
Completely delete the current cursor line.
Duplicates the cursor line, inserting a copy of the line immediately after the cursor line.
Deletes the line at the cursor if it is empty or contains only whitespace.



Encoding dialog box

This dialog box is used to select a file-encoding.

Remark The target file name is displayed on the title bar.

Figure A.3 Encoding Dialog Box

	main.c - Encoding	
(1)-	Available encodings: 1146: IBM EBCDIC (UK-Euro) 1147: IBM EBCDIC (France-Euro) 1148: IBM EBCDIC (International-Euro) 1149: IBM EBCDIC (Icelandic-Euro) 1200: Unicode 1201: Unicode (Big-Endian) 1250: Central European (Windows) 1251: Cyrillic (Windows) 1252: Western European (Windows)	
unction but-	OK Cancel	Help

The following items are explained here.

- [How to open]

[F

- [Description of each area]
- [Function buttons]

[How to open]

- From the [File] menu, open the Open File dialog box by selecting [Open with Encoding...], and then click the [Open] button in the dialog box.

[Description of each area]

 (1) [Available encodings] Select the encoding to be set from the drop-down list. All code-pages and encodings supported by the OS are displayed in alphabetical order. Note that the same encoding and encoding which are not supported by the current OS will not be displayed. The default file-encoding in the [General - Text Editor] category of the Option dialog box is selected by default.

[Function buttons]

Button	Function
ОК	Opens the selected file in the Open File dialog box using a selected file encoding.
Cancel	Not open the selected file in the Open File dialog box and closes this dialog box.
Help	Displays the help for this dialog box.



Go to Line dialog box

This dialog box is used to move the caret to a specified source line.

Figure A.4 Go to Line Dialog Box

		Go to Line 🛛 🔀
	(1)-	Line number (1 - 133): 84
[Function	-[OK Cancel <u>H</u> elp

The following items are explained here.

- [How to open]
- [Description of each area]
- [Function buttons]

[How to open]

- From the [Edit] menu, select [Go To...].
- On the Editor panel, select [Go To...] from the context menu.

[Description of each area]

(1) [Line number (valid line range)]
 "(valid line range)" shows the range of valid lines in the current file.
 Directly enter a decimal value as the number of the line you want to move the caret to.
 By default, the number of the line where the caret is currently located in the Editor panel is displayed.

[Function buttons]

Button	Function
ОК	Places the caret at the start of the specified source line.
Cancel	Cancels the jump and closes this dialog box.
Help	Displays the help for this dialog box.



Print Preview window

This window is used to preview the file currently being displayed in the Editor panel before printing.

- Remark This window can be zoomed in and out by moving the mouse wheel forward or backward while holding down the [Ctrl] key.
- Figure A.5 Print Preview Window

	🚨 Print Preview	K
[Toolbar]		-
	void main(void)	^
	/* Start user code. Do not edit comment ge:	
	<pre>TMP0_Start(); TMP1_Start();</pre>	
(1)—	AD_Start ();	
	while (1U)	
	func1(); func2();	
	}	~
_		

The following items are explained here.

- [How to open]
- [Description of each area]
- [Toolbar]
- [Context menu]

[How to open]

- Focus the Editor panel, and then select [Print Preview] from the [File] menu.

[Description of each area]

(1) Preview area

This window displays a form showing a preview of how and what is printed.

The file name (fully qualified path) and the page number are displayed at the page header and page footer. The display differs according to whether the debug tool is or is not connected, and when it is connected, to whether the display is in normal display mode or mixed display mode. Note, however, that columns that are hidden on the Editor panel are not displayed (these columns are not printed).

When the outline setting is in used and the collapsed section mark of an outline (see "(a) Code outlining") is displayed in a print preview, the lines in the collapsed section are also displayed.

[Toolbar]

4

Opens the Print dialog box provided by Windows to print the current Editor panel as shown by the print preview form.



\square	Copies the selection into the clipboard.
Ð	Increases the size of the content.
Q	Decreases the size of the content.
-	Displays the preview at 100-percent zoom (default).
Į	Fits the preview to the width of this window.
	Displays the whole page.
	Displays facing pages.

[Context menu]

Increase Zoom	Increases the size of the content.
Decrease Zoom	Decreases the size of the content.



Open File dialog box

This dialog box is used to open a file.

Figure A 6	Open File Dialog Box
i igule A.0	Open i lie Dialog Dox

	Open File							? 🛛
$-\Box$	Look jn:	🚞 sample		*	G	B 🖻	• 📰 -	
	My Recent Documents	CefaultBuild						
	Desktop							
	My Documents							
	My Computer							
		►File <u>n</u> ame:				-	/	<u>O</u> pen
-	My Network	Files of <u>type</u> :	Project File(*.mtpj)			1	/	Cancel

The following items are explained here.

- [How to open]
- [Description of each area]
- [Function buttons]

[How to open]

- From the [File] menu, select [Open File...] or [Open with Encoding...].

[Description of each area]

- (1) [Look in] area Select the folder that the file you want to open exists. When you first open this dialog box, the folder is set to "C:\Documents and Settings \user-name\My Documents". The second and subsequent times, this defaults to the last folder that was selected.
- (2) List of files areaFile list that matches to the selections in [Look in] and [Files of type] is shown.
- (3) [File name] area Specify the file name that you want to open.



(4)

[Files of type] area Select the type of the file you want to open.

All files (*.*)	All formats
Project File(*.mtpj)	Project file
Project File for CubeSuite(*.cspj)	Project file for CubeSuite
Workspace File for HEW(*.hws)	Workspace file for HEW
Project File for HEW(*.hwp)	Project file for HEW
Workspace File for PM+(*.prw)	Workspace file for PM+
Project File for PM+(*.prj)	Project file for PM+
C source file (*.c)	C language source file
C++ source file (*.cpp; *.cc; *.cp)	C++ language source file
Header file (*.h; *.hpp; *.inc)	Header file
Assemble file (*.src)	Assembler source file
Map file (*.map; *.lbp)	Map file
Hex file (*.hex)	Hex file
Assemble list file (*.lst)	Assemble list file
S record file (*.mot)	S record file
Text file (*.txt)	Text format

[Function buttons]

Button	Function		
Open	 When this dialog box is opened by [Open File] from the [File] menu Opens the specified file. 		
	- When this dialog box is opened by [Open with Encoding] from the [File] menu Opens the Encoding dialog box.		
Cancel	Closes this dialog box.		



Revision Record

Rev.	Date	Description			Description		
		Page	Summary				
Rev.1.00	May. 20, 2014	-	First Edition issued				

CubeSuite+ V2.02.00 User's Manual: RX Coding				
Publication Date:	Rev.1.00	May. 20, 2014		
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. 2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A. Tel: +1-408-588-6000, Fax: +1-408-588-6130 Renesas Electronics Canada Limited 1011 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada Tel: +1-905-898-5441, Fax: +1-905-898-3220 Renesas Electronics Europe Limited Dukes Meadow, Milboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K Tel: +44-1828-585-100, Fax: +44-1628-585-900 Renesas Electronics (China) Co., Ltd. Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China Tel: +49-21-16503-0, Fax: +49-211-6503-1327 Renesas Electronics (China) Co., Ltd. Nor Market Alexan, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China Tel: +48-21-16503-0, Fax: +48-21-1628-57679 Renesas Electronics (Shanghai) Co., Ltd. Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333 Tel: +48-21-2226-0888, Fax: +485-21-2226-0999 Renesas Electronics (Shanghai) Co., Ltd. Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333 Tel: +485-21-2226-5688, Fax: +485-2486-20229044 Renesas Electronics Taiwan Co., Ltd. Unit 305, Fu Shing North Road, Taipei 10543, Taiwan Tel: +485-2155-5688, Fax: +485-2486-20229044 Renesas Electronics Singapore Pte. Ltd. 80 Bendemer Road, Unit 90-62 Hythy Innovation Centre, Singapore 339949 Tel: +485-2415-9900, Fax: +485-2481-230-300 Renesas Electronics Malaysia Ameorp, Ameorp Trade Centre, No. 18, Jin Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia Tel: +482-24558-3930, Fax: +685-2419-200, Korea Tel: +482-24558-3737, Fax: +482-24559-5910

> © 2014 Renesas Electronics Corporation and Renesas Solutions Corp. All rights reserved. Colophon 3.0

CubeSuite+ V2.02.00

