

# H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor

Compiler Package Ver.6.01 User's Manual

Renesas Microcomputer Development Environment System

Rev.1.00

Revision Date: Jan. 12, 2005

Renesas Technology  
[www.renesas.com](http://www.renesas.com)

User's Manual



## Keep safety first in your circuit designs!

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

## Notes regarding these materials

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



This manual describes the facilities and operating procedures for the H8S, H8/300 series C/C++ compiler (hereinafter H8S, H8/300 compiler or simply the compiler). The compiler translates source programs written in C/C++ into object programs and load modules for Renesas H8SX series, H8S/2600 series, H8S/2000 series, H8/300H series, H8/300 series, and H8/300L series microcomputers. Please read this H8S, H8/300 Series C/C++ Compiler User's Manual before using the compiler to fully understand the system.

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

### Symbols Used in This Manual

Symbol	Explanation
< >	Indicates an item to be specified.
[ ]	Indicates an item that can be omitted.
...	Indicates that the preceding item can be repeated.
Δ	Indicates one or more blanks.
(RET)	Indicates the carriage return key (return key).
	Indicates that one of the items must be selected.
(CNTL)	Indicates that the control key should be held down while pressing the key that follows.

This manual is intended for UNIX<sup>\*1</sup>, Microsoft® Windows® 98 operating system, Microsoft® Windows® Millennium Edition operating system, Microsoft® Windows NT® operating system, Microsoft® Windows® 2000 operating system, Microsoft® Windows® XP operating system<sup>\*2</sup> and other compatible systems. In this document, the compiler functioning on a UNIX system is referred to as the UNIX version. The compiler operating in IBM PC<sup>\*3</sup> and other compatible computers are referred to as the PC version.

- Notes:
1. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.
  2. Microsoft®, Windows®, and WindowsNT® are registered trademarks of Microsoft Corporation in the United States and/or other countries.
  3. IBM is a registered trademark of International Business Machines Corporation.

# Contents

Section 1	Overview .....	1
1.1	Procedures for Developing Programs .....	1
1.2	Compiler .....	3
1.3	Assembler .....	3
1.4	Optimizing Linkage Editor .....	4
1.5	Prelinker .....	4
1.6	Standard Library Generator .....	4
1.7	Stack Analysis Tool .....	5
1.8	Format Converter .....	5
Section 2	C/C++ Compiler Operating Method.....	7
2.1	Command Line Format .....	7
2.2	Interpretation of Options.....	7
2.2.1	Source Options.....	8
2.2.2	Object Options .....	12
2.2.3	List Options.....	22
2.2.4	Optimize Options.....	25
2.2.5	Other Options.....	41
2.2.6	CPU Options .....	50
2.2.7	Options Other Than Above.....	60
Section 3	Assembler Options .....	65
3.1	Command Line Format .....	65
3.2	List of Options .....	65
3.2.1	Source Options.....	66
3.2.2	Object Options .....	71
3.2.3	List Options.....	78
3.2.4	Tuning Options .....	84
3.2.5	Other Options.....	86
3.2.6	CPU Options .....	87
3.2.7	Options Other Than Above.....	93
Section 4	Optimizing Linkage Editor Options .....	101
4.1	Option Specifications.....	101
4.1.1	Command Line Format .....	101
4.1.2	Subcommand File Format.....	101
4.2	List of Options .....	101
4.2.1	Input Options .....	102
4.2.2	Output Options.....	107

4.2.3	List Options.....	117
4.2.4	Optimize Options .....	119
4.2.5	Section Options .....	125
4.2.6	Verify Options .....	127
4.2.7	Other Options.....	129
4.2.8	Subcommand File Option .....	137
4.2.9	CPU Option.....	138
4.2.10	Options Other Than Above.....	139
<b>Section 5</b>	<b>Standard Library Generator Operating Method.....</b>	<b>141</b>
5.1	Comand Line Format .....	141
5.2	Option Descriptions .....	141
5.2.1	Additional Options.....	142
5.2.2	Options Unavailable for Standard Library Generator .....	145
5.2.3	Notes on Specifying Options .....	146
<b>Section 6</b>	<b>Operating Stack Analysis Tool.....</b>	<b>147</b>
6.1	Overview.....	147
6.2	Starting the Stack Analysis Tool.....	147
<b>Section 7</b>	<b>Environment Variables .....</b>	<b>149</b>
7.1	Environment Variables List .....	149
7.2	Compiler Implicit Declaration .....	153
<b>Section 8</b>	<b>File Specifications.....</b>	<b>155</b>
8.1	Naming Files .....	155
8.2	Compiler Listings.....	157
8.2.1	Structure of Compiler Listings.....	157
8.2.2	Source Listing .....	158
8.2.3	Error Information .....	160
8.2.4	Symbol Allocation Information .....	161
8.2.5	Object Information.....	164
8.2.6	Statistics Information .....	166
8.3	Assembler Listings.....	167
8.3.1	Structure of Assembler Listings.....	167
8.3.2	Source Listing .....	167
8.3.3	Cross Reference Listing .....	169
8.3.4	Section Information Listing .....	171
8.4	Linkage Listings.....	171
8.4.1	Structure of Linkage Listing .....	172
8.4.2	Option Information .....	173
8.4.3	Error Information .....	173
8.4.4	Linkage Map Information .....	174

8.4.5	Symbol Information .....	174
8.4.6	Symbol Deletion Optimization Information .....	176
8.4.7	Variable Access Optimization Symbol Information .....	176
8.4.8	Function Access Optimization Symbol Information.....	178
8.4.9	Cross-Reference Information.....	179
8.5	Library Listings.....	180
8.5.1	Structure of Library Listing .....	180
8.5.2	Option Information .....	181
8.5.3	Error Information.....	182
8.5.4	Library Information .....	182
8.5.5	Module, Section, and Symbol Information within Library .....	183
<b>Section 9</b>	<b>Programming .....</b>	<b>185</b>
9.1	Program Structure .....	185
9.1.1	Sections.....	185
9.1.2	C/C++ Program Sections .....	185
9.1.3	Assembly Program Sections .....	189
9.1.4	Linking Sections .....	191
9.2	Creation of Initial Setting Programs .....	195
9.2.1	Memory Allocation.....	195
9.2.2	Execution Environment Settings.....	205
9.3	Linking C/C++ Programs and Assembly Programs .....	244
9.3.1	Method for Mutual Referencing of External Names.....	244
9.3.2	Function Calling Interface .....	246
9.3.3	Examples of Parameter Assignment .....	257
9.3.4	Using the Registers and Stack Area.....	267
9.4	Important Information on Program Creation .....	272
9.4.1	Important Information on Program Coding .....	272
9.4.2	Important Information on Compiling a C Program with the C++ Compiler.....	275
9.4.3	Important Information on Program Development.....	276
<b>Section 10</b>	<b>C/C++ Language Specifications.....</b>	<b>279</b>
10.1	Language Specifications .....	279
10.1.1	Compiler Specifications.....	279
10.1.2	Internal Data Representation.....	288
10.1.3	Floating-Point Number Specifications .....	302
10.1.4	Operator Evaluation Order.....	310
10.2	Extended Functions.....	311
10.2.1	#pragma Extension Specifiers and Keywords.....	311
10.2.2	Section Address Operator .....	359
10.2.3	Intrinsic Functions .....	361
10.3	C/C++ Libraries .....	390
10.3.1	Standard C Libraries .....	390



10.3.2	Embedded C++ Class Libraries .....	533
10.3.3	Reentrant Library .....	620
10.3.4	Unsupported Libraries.....	625
<b>Section 11</b>	<b>Assembly Specifications .....</b>	<b>627</b>
11.1	Program Elements .....	627
11.1.1	Source Statements .....	627
11.1.2	Reserved Words .....	631
11.1.3	Symbols .....	631
11.1.4	Constants .....	635
11.1.5	Location Counter .....	637
11.1.6	Expressions .....	638
11.1.7	String Literal .....	647
11.1.8	Local Label .....	648
11.2	Executable Instructions .....	650
11.2.1	Overview of Executable Instructions .....	650
11.2.2	Notes on Executable Instructions .....	652
11.3	Assembler Directives .....	674
11.4	File Inclusion Function .....	749
11.5	Conditional Assembly Function.....	752
11.5.1	Overview of the Conditional Assembly Function.....	752
11.5.2	Conditional Assembly Directives .....	758
11.6	Macro Function.....	774
11.6.1	Overview of the Macro Function .....	774
11.6.2	Macro Function Directives.....	776
11.6.3	Macro Body .....	780
11.6.4	Macro Call .....	784
11.6.5	String Literal Manipulation Functions .....	786
11.7	Overview of Structured Assembly .....	790
11.7.1	Notes on Structured Assembly .....	791
11.7.2	Structured Assembly Directives.....	792
<b>Section 12</b>	<b>Compiler Error Messages .....</b>	<b>815</b>
12.1	Error Format and Error Levels .....	815
12.2	Error Messages.....	815
12.3	C Library Function Error Messages .....	882
<b>Section 13</b>	<b>Assembler Error Messages .....</b>	<b>885</b>
13.1	Error Message Format and Error Levels .....	885
13.2	Error Messages.....	885
<b>Section 14</b>	<b>Error Messages for the Optimizing Linkage Editor.....</b>	<b>903</b>
14.1	Error Format and Error Levels .....	903

14.2 List of Messages .....	903
<b>Section 15 Error Messages for the Standard Library Generator and Format Converter 917</b>	
15.1 Error Format and Error Levels.....	917
15.2 List of Messages .....	917
<b>Section 16 Limitations.....</b>	<b>921</b>
16.1 Limitations of the Compiler.....	921
16.2 Limitations of the Assembler.....	924
<b>Section 17 Supporting AE5 Features .....</b>	<b>925</b>
17.1 Compiler Functions.....	925
17.1.1 Overview.....	925
17.1.2 Compiler Options.....	925
17.1.3 Intrinsic Functions .....	927
17.2 Assembler Functions.....	930
<b>Section 18 Notes on Version Upgrade .....</b>	<b>933</b>
18.1 Notes on Version Upgrade.....	933
18.1.1 Guaranteed Program Operation .....	933
18.1.2 Compatibility with the Earlier Version.....	934
18.1.3 Command-line Interface .....	937
18.1.4 Provided Contents.....	940
18.1.5 List File Specification.....	941
18.2 Additions and Improvements.....	941
18.2.1 Common Additions and Improvements .....	941
18.2.2 Added and Improved Compiler Features .....	942
18.2.3 Added and Improved Features for the Assembler.....	953
18.2.4 Added and Improved Features for the Optimizing Linkage Editor.....	954
18.3 Operating Format Converter .....	956
18.3.1 Object File Format .....	956
18.3.2 Compatibility with Earlier Versions .....	956
18.3.3 Command Line Format .....	957
18.3.4 Interpretation of Options.....	957
<b>Section 19 Appendix .....</b>	<b>961</b>
19.1 S-Type and HEX File Format .....	961
19.1.1 S-Type File Format.....	961
19.1.2 HEX File Format .....	963
19.2 ASCII Code List .....	965
19.3 Access Range of Short Absolute Addresses .....	966

Index ..... 967



# Section 1 Overview

## 1.1 Procedures for Developing Programs

Figure 1.1 shows the procedures for developing programs. The shaded parts show software provided in the Renesas C/C++ Compiler Package for H8, H8S and H8SX family.

The C/C++ compiler, assembler, optimizing linkage editor, standard library generator, stack analysis tool, and format converter are explained in this manual.



Outlines of the C/C++ compiler, assembler, optimizing linkage editor, prelinker, standard library generator, stack analysis tool, and format converter are given in the following sections.

## 1.2 Compiler

The H8S, H8/300 series C/C++ compiler (hereinafter referred to as compiler) is software that takes source programs written in C or C++ language as inputs, and produces relocatable object programs or assembly source programs for the H8S, H8/300 series microcomputers.

Features of this compiler are as follows:

1. Generates an object program that can be written to ROM for installation in a user system.
2. Supports an optimization that improves the speed of execution of object programs and minimizes program size.
3. Supports extended features and options to take advantage of CPU's features such as short absolute addressing mode and indirect addressing mode.
4. Supports the C and C++ programming languages.
5. Supports features that are essential for the programming of embedded programs but are not standards in the C and C++ languages as extended features. Such features include interrupt functions and descriptions of system instructions.
6. Supports output of debugging information to enable C/C++ source-level debugging by the debugger.
7. Either an assembly source program or a relocatable object program can be selected for output.
8. Supports output of an inter-module optimization information used by the optimizing linkage editor.

## 1.3 Assembler

The H8S, H8/300 series assembler (hereinafter referred to as assembler) takes source programs written in assembly language, and outputs relocatable object programs for the H8S, H8/300 series microcomputers.

Features of this assembler are as follows:

1. Enables the efficient writing of source programs by providing the preprocessor functions listed below:
  - File include function
  - Conditional assembly function
  - Macro function
  - Structured assembly function
2. The mnemonics for execution instructions and assembly directives conform to the naming rules laid out in the IEEE-694 specifications, and the system is uniform.

## 1.4 Optimizing Linkage Editor

The optimizing linkage editor is software that takes multiple object programs output by the compiler or assembler and produces a load module or a library file.

Features of this optimizing linkage editor are as follows:

1. Optimization can be applied to a set of several object files, depending on memory allocation and relations among function calls which cannot be optimized by the compiler.
2. Any of the following five types of load modules can be selected for output:
  - Relocatable ELF format
  - Absolute ELF format
  - S-type format
  - HEX format
  - Binary format
3. Generates and edits library files.
4. Outputs symbol reference count list.
5. Deletes debugging information from library and load module files.
6. Specifies the output of a stack information file for use by the stack analysis tool.

## 1.5 Prelinker

The prelinker is called from the optimizing linkage editor. When a C++ program template or runtime type information is used, the prelinker calls the compiler and makes it generate the necessary object files. When neither a C++ program template nor the runtime type information is used, the speed of linkage can be improved by specifying the **noprelink** option for the optimizing linkage editor.

## 1.6 Standard Library Generator

The H8S, H8/300 series standard library generator (hereinafter referred to as the standard library generator) is a software system for the reconfiguration of standard library files provided, using user-specified options.

The standard library functions provided with the compiler include the standard set of C library functions, a set of C++ class library functions for embedded systems, and a set of runtime routines (arithmetic operations that are necessary for the execution of a program). In some cases, runtime routines will be necessary, even though the use of library functions in source programs has not been specified.



## **1.7 Stack Analysis Tool**

The stack analysis tool is software that takes the stack information file that is output by the optimizing linkage editor and calculates the size of the stack that will be used by C/C++ programs.

## **1.8 Format Converter**

The ELF/DWARF format converter (hereinafter referred to as format converter) takes object files and library files that have been output by an earlier version of the compiler or assembler and converts them to the ELF format. It can also take an ELF-format absolute load module and convert it to the output format of an earlier version of the linkage editor.



# Section 2 C/C++ Compiler Operating Method

## 2.1 Command Line Format

The format of the command line to initiate the compiler is as follows:

```
ch38[Δ<option>...][Δ<file name>[Δ<option>...] ...]  
    <option>:-<option>[=<suboption>][,...]
```

## 2.2 Interpretation of Options

In the command line format, uppercase letters indicate the abbreviation and characters underlined indicate the defaults setting.

The dialog menus of the HEW is shown in the form of

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

The order of options correspond to that of the tabs and the categories in the HEW.

## 2.2.1 Source Options

**Table 2.1 Source Options**

Item	Command Line Format	Dialog Menu	Specification
Include file directory	Include = <path name>[,...]	C/C++ <Source> [Show entries for :] [Include file directories]	Specifies include-file include path name.
Default include file	PREInclude = <file name>[,...]	C/C++ <Source> [Show entries for :] [Preinclude files]	Includes the specified files at the head of compiling units.
Macro name definition	DEFine = <sub>[,...] <sub>: <macro name> [=<string literal>]	C/C++ <Source> [Show entries for :] [Defines]	Defines <string literal> as <macro name>.
Information message output control	Message <u>NOMessage</u> [= <error code> [-<error code>][,...]]	C/C++ <Source> [Show entries for :] [Messages] [Display information level messages]	Outputs information message. Does not output information message (error number and range can be specified).
Inter-file inline expansion directory specification	FILE_INLINE_PATH = <path name>[,...]	C/C++ <Source> [Show entries for :] [File inline path]	Specifies the path name where obtains a file that has function definitions to be expanded as inline functions.

## Include: Include File Directory

C/C++ <Source>[Show entries for :][Include file directories]

- Command Line Format

Include = <path name>[,...]

- Description

Specifies the name of the path where the include file is stored.

Two or more path names can be specified by separating them with a comma (,).

System include files are retrieved in the order of **include** specification directory and the environment variable CH38 specification directory. User include files are retrieved in the order of the current directory, **include** specification directory, and the environment variable CH38 specification directory.

- Example

```
ch38 -include=c:\usr\inc,c:\usr\CH38 test.c
```

Directories c:\usr\inc and c:\usr\CH38 are retrieved as include file paths.

## PREInclude: Default Include File

C/C++ <Source>[Show entries for :][Preinclude files]

- Command Line Format

PREInclude = <file name>[,...]

- Description

Includes the specified file contents at the head of the compiling unit. Two or more path names can be specified by separating them with a comma (,).

- Example

```
ch38 -preinclude=a.h test.c
```

— Contents of <test.c>

```
int a;  
main(){...}
```

— Interpretation at compilation

```
#include "a.h"  
int a;  
main(){...}
```

## DEFine: Macro Name Definition

C/C++ <Source>[Show entries for :][Defines]

- Command Line Format

DEFine = <sub> [,...]

<sub>: <macro name> [= <string literal>]

- Description

This option is the same as #define written in the C/C++ source file.

When <macro name>=<string literal> is specified, <string literal> is defined as a macro name.

When only <macro name> is specified for a suboption, the macro name is regarded as defined.

<string literal> allows name or constant integer.

## Message, NOMessage: Information Message

C/C++ <Source>[Show entries for :][Messages][Display information level messages]

- Command Line Format

Message

NOMessage [= <error code> [- <error code>] [,...]]

- Description

Specifies whether to output information-level messages.

If **message** is specified, the compiler outputs information-level messages.

If **nomessage** alone is specified, the compiler does not output any information-level messages.

If an error code is specified for the suboption, display of messages of the specified codes is disabled. The range of error messages to be disabled can also be specified for the suboption by using a hyphen (-):

<error code> - <error code>.

When this option is not specified, the compiler assumes that **nomessage** is specified.

- Example

ch38 -nomessage=5,300-306 test.c

Information-level message codes C0005 and C0300 to C0306 will not be displayed.

- Remarks

An <error code> allows Warning or Information code.

The Ver. 4.0 or earlier version of the compiler validates only the last specification of **message** or **nomessage** options when such options are specified more than once. This version, Ver. 6.0, or later suppresses output of the union of messages specified by the **nomessage** options.

## FILE\_INLINE\_PATH: Inter-file Inline Expansion Directory Specification

C/C++ <Source>[Show entries for :][File inline path]

- Command Line Format

FILE\_INLINE\_PATH = <path name> [...]

- Description

Specifies the name of the path where a file for inter-file inline expansion is stored.

Two or more path names can be specified by separating them with a comma (,).

Files for inter-file inline expansion are retrieved in the order of the **file\_inline\_path** option specification directory and the current directory.

- Example

ch38 -file\_inline\_path=c:\usr\file -file\_inline=test2.c test.c

A directory “c:\usr\file” is as inter-inline expansion searching directory and the compiler try to find the “test2.c” as “file\_inline” option.

- Remarks

This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

## 2.2.2 Object Options

**Table 2.2 Object Options**

Item	Command Line Format	Dialog Menu	Specification
Pre-processor expansion	PREProcessor [= <file name>]	C/C++ <Object> [Output file type :] [Preprocessed source file]	Outputs source program after preprocessor expansion.
Object type	Code =  { <u>Machinecode</u>   <u>Asmcode</u> }	C/C++ <Object> [Output file type :] [Machine code] [Assembly source code]	Outputs machine code program. Outputs assembly-source program.
Debugging information	DEBug <u>NODEBug</u>	C/C++ <Object> [Generate debug information]	Outputs debug information Not output debug information
Section name	SEction = <sub>[,...] <sub>:{ Program=<section name>   Const=<section name>   Data=<section name>   Bss=<section name> }	C/C++ <Object> [Section :] [Program section (P)] [Const section (C)] [Data section (D)] [Uninitialized data section (B)]	Program area section name Constant area section name Initialized data area section name Non-initialized data area section name
Area of string literal to be output	STring = { <u>Const</u>    <u>Data</u> }	C/C++ <Object> [Store string data in :]	Outputs string literal to constant section (C). Outputs string literal to initialized data section (D).
Operation size expanded interpretation	CPUEExpand [=V6]  <u>NOCPUEExpand</u>	C/C++ <Object> [Mul/Div operation specification]	Multiplication and division are code-generated by the CPU instruction specifications. Multiplication and division are code-generated based on the ANSI C-language specification.
Object file output specification	<u>ObJect</u> [= <file name>]  NOObJect	C/C++ <Object> [Output directory :]	Outputs an object file. Not output an object file.



**Table 2.2 Object Options (cont)**

Item	Command Line Format	Dialog Menu	Specification
Template instance generation	Template={ None   Static   Used   ALI   <u>AUto</u> }	C/C++ <Object> [Template :]	Not generate instances.  Generates instances as internal linkage only for referenced templates Generates instances as external linkage only for referenced templates. Generates instances for templates defined or referenced. Generates instances at linkage
Boundary alignment value and disable of boundary alignment	<u>ALign</u> [=4] NOALign	C/C++ <Object> [Group by alignment :]	Modifies allocation order by the boundary alignment.  Allocates the variables in the order of declaration.
Compatibility of output object code	LEgacy=v4	C/C++ <Object> [Ver.4.0 Optimization technology generation:]	Output objects generated by Ver.4.0 optimization technology of H8S

**PREProcessor: Preprocessor Expansion**

C/C++ <Object>[Output file type :][Preprocessed source file]

- Command Line Format  
PREProcessor [= <file name>]

- Description

Outputs a source program processed by the preprocessor.

If no <file name> is specified, an output file with the same file name as the source file and with a standard extension is created. The standard extension after C compilation is “p” (if the input source program is written in C), and that after C++ compilation is “pp” (if the input source program is written in C++).

When **preprocessor** is specified, no object file is output by the compiler.

- Remarks

When **preprocessor** is specified, the following options become invalid:

code, object, outcode, debug, pack, string, show=object, statistics, allocation, section, optimize, speed, goptimize, bytenum, volatile, regexpansion, cmncode, case, indirect, abs8, abs16, cpuexpand, eepmov, regparam, stack, align/noalign, structreg, longreg, macsave, bit\_order, ptr16, opt\_range, del\_vacant\_loop, max\_unroll, infinite\_loop, global\_alloc, struct\_alloc, const\_var\_propagate, library, volatile\_loop, sbr, legacy=v4, scope, noscope, file\_inline, file\_inline\_path, enable\_register, strict\_ansi and cpuexpand=v6.

## Code: Object Type

C/C++ <Object>[Output file type :] [Machine code] [Assembly source code]

- Command Line Format

Code = { Machinecode | Asmcode }

- Description

Specifies an object file output type.

When **code=machinecode** is specified, a relocatable object program (in machine code) is generated.

When **code=asmcode** is specified, an assembly source program is generated.

Within the assembly program, stack information usage by all functions is reflected by .stack directives.

When this option is not specified, the compiler assumes that **code=machinecode** is specified.

- Remarks

When **code=asmcode** is specified, **show=object** or **goptimize** becomes invalid.

## DEBug, NODEBug: Debugging Information

C/C++ <Object>[Generate debug information]

- Command Line Format

DEBug

NODEBug

- Description

Specifies whether to output the information necessary for source-level debugging into the object file.

This option is valid regardless of the optimization option specified.

When **nodebug** is specified, no debugging information will be output to the object file.

If this option is not specified, the compiler will assume **nodebug** is specified.

## SSection: Section Name

C/C++ <Object>[Section :] [Program section (P)] [Const section (C)] [Data section (D)]  
[Uninitialized data section (B)]

- Command Line Format

SSection = <sub> [,...]

<sub>: { Program=<section name> |  
Const=<section name> |  
Data=<section name> |  
Bss=<section name> }

- Description

Specifies the section name of an object program.

**section=program=<section name>** specifies the section name in of the program area.

**section=const=<section name>** specifies the section name in of the constant area.

**section=data=<section name>** specifies the section name in of the initialized data area.

**section=bss=<section name>** specifies the section name in of the non-initialized data area.

The <section name> must consist of alphabetic, numerics, underscore (\_) or dollar sign (\$) except that the first character must not be numeric. The section name must be specified within 8192 characters.

The default section names are as follows: P for the program area section, C for the constant area section, D for the initialized data area section, and B for the non-initialized data area section.

- Remarks

For details on programs and section names, refer to section 9.1, Program Structure. The same section name cannot be specified for different areas of the section. Changing the section name of P, C, B or D into S by **section** causes a warning error because S is the reserved name for the stack area.

## SString: String Literal Output Area

C/C++ <Object>[Store string data in :]

- Command Line Format

SString = { Const | Data }

- Description

Specifies the destination where string literal is output.

When **string=const** is specified, the compiler outputs the string literal to the constant area.

When **string=data** is specified, the compiler outputs the string literal to the initialized data area.

The string literal output to the initialized data area can be modified during program execution; however, the initialized data area must be allocated in both ROM and RAM in order to transfer the string literal to RAM from ROM at the beginning of program execution. For details on the initial settings of the initialized data area or on memory allocation, refer to section 9.2.1 Memory Allocation.

When this option is not specified, the compiler assumes that **string=const** is specified.

## **CPUExpand, NOCPUExpand: Operation Size Expanded Interpretation**

C/C++ <Object>[Mul/Div operation specification]

- Command Line Format

CPUExpand [=V6]

NOCPUExpand

- Description

**cpuexpand** generates multiplication and division code for variables by deviating from the ANSI C-language standard.

Specifying **cpuexpand=v6** makes Ver.6.0 cpuexpand specification when output code is generated by Ver.4.0 optimization technology.

With this sub-option, generated codes are affected by the following C-source descriptions.

- (a) signed long = signed int << Constant
- (b) signed long = unsigned int << Constant
- (c) unsigned long = signed int << Constant
- (d) unsigned long = unsigned int << Constant
- (e) signed int = (signed int << Constant) / signed int
- (f) signed int = (unsigned int << Constant) / signed int
- (g) signed int = (unsigned int << Constant) / unsigned int
- (h) unsigned int = (signed int << Constant) / signed int
- (i) unsigned int = (unsigned int << Constant) / signed int
- (j) unsigned int = (unsigned int << Constant) / unsigned int

When **nocpuexpand** is specified, the compiler generates multiplication and division code conforming to the ANSI C-language standard.

When this option is not specified, the compiler assumes that **nocpuexpand** is specified.

- Remarks

When **cpuexpand** and **cpuexpand=V6** is specified, the operation specifications exceed the range guaranteed by the C language specifications, and the result may be different from that obtained when **nocpuexpand** is specified.

Table 2.3 shows examples of multiplication and division code generated by specifying this option.

**Table 2.3    cpuexpand Option Specifications**

Operation	Operation Size of us1*us2 (for H8S/2600)	
	cpuexpand Is Specified	nocpuexpand Is Specified
unsigned short us1,us2; unsigned long ul; ul=us1*us2;	<p>The intermediate result is held as unsigned long.*</p> <p>Output example:</p> <pre> MOV.W    @_us1,Rd MOV.W    @_us2,Rs MULXU.W  Rs,ERd MOV.L    ERd,@_ul </pre> <p>4-byte result of us1*us2 is assigned to ul.</p>	<p>Calculated as unsigned short.</p> <p>Output example:</p> <pre> MOV.W    @_us1,Rd MOV.W    @_us2,Rs MULXU.W  Rs,ERd EXTU.L   ERd MOV.L    ERd,@_ul </pre> <p>Low-order two bytes of us1*us2 result are zero-extended and assigned to ul.</p>
unsigned short us1,us2,us3; unsigned short us; us=us1*us2/us3;	<p>The intermediate result is held as unsigned long.*</p> <p>Output example:</p> <pre> MOV.W    @_us1,Rd MOV.W    @_us2,Rs MULXU.W  Rs,ERd MOV.W    @_us3,Rs DIVXU.W  Rs,ERd MOV.W    Rd,@_us </pre> <p>4-byte result of us1*us2 is used as the dividend.</p>	<p>Calculated as unsigned short.</p> <p>Output example:</p> <pre> MOV.W    @_us1,Rd MOV.W    @_us2,Rs MULXU.W  Rs,ERd EXTU.L   ERd MOV.W    @_us3,Rs DIVXU.W  Rs,ERd MOV.W    Rd,@_us </pre> <p>Low-order two bytes of us1*us2 result are zero-extended and used as the dividend.</p>

**Note:** The intermediate 4-byte result of a multiplication of two 2-byte data is used as it is if the result is assigned to or converted to a 4-byte object, or is divided by a 2-byte divisor.

**cpuexpand=V6** is valid only when the CPU type is H8S and **legacy=v4** has been specified or CPU type is H8/300 and H8/300H.

## Object, NOObject: Object File Output

C/C++ <Object>[Output directory :]

- Command Line Format

Object [= <object file name>]

NOObject

- Description

Specifies whether or not to output an object file.

When **noobject** is specified, no object file is output.

If <object file name> is not specified in **object**, the object file name becomes the same as that of the source file and the extension becomes “obj” for a relocatable object program and “src” for an assembly source program, which is determined by the **code** option.

When this option is not specified, the compiler assumes that **object** is specified.

- Remarks

When **noobject** is specified, the following options become invalid:

outcode, debug, pack, string, show=object, statistics, allocation, section, optimize, speed, goptimize, byteenum, volatile, regexpansion, cmncode, case, indirect, abs8, abs16, cpuexpand, eepmov, regparam, stack, align/noalign, structreg, longreg, macsave, bit\_order, ptr16, opt\_range, del\_vacant\_loop, max\_unroll, infinite\_loop, global\_alloc, struct\_alloc, const\_var\_propagate, library, volatile\_loop, sbr, legacy=v4, scope, noscope, file\_inline, file\_inline\_path, enable\_register, strict\_ansi and cpuexpand=v6.

## Template: Template Instance Generation

C/C++ <Object>[Template :]

- Command Line Format

Template = { None

| Static

| Used | All | Auto }

- Description

Specifies the condition to generate template instances.

When **template=none** is specified, instances are not generated.

When **template=static** is specified, instances of templates referenced in the compiling unit are generated. However, generated functions contain the internal linkage.

When **template=used** is specified, instances of templates referenced in the compiling unit are generated. However, generated functions contain the external linkage.

When **template=all** is specified, instances of all templates defined or referenced in the compiling unit are generated.

When **template=auto** is specified, instances needed at linkage are generated.

When this option is not specified, the compiler assumes that **template=auto** is specified.

- Remarks

When a code = asmcode is specified, **template=static** is always valid.

## **ALign, NOALign: Boundary Alignment Value and Disable of Boundary Alignment**

C/C++ <Object>[Group by alignment :]

- Command Line Format

ALign [=4]

NOALign

- Description

The **noalign** option allocates defined variables in the order of declaration.

The **align** option relocates variables so as to reduce space by boundary alignment. When the relocation is performed, generally the empty area is reduced and the object size is also reduced.

The **align=4** option divides a data section into a 4-byte boundary alignment section, a 2-byte boundary alignment section and a 1-byte boundary alignment section. A datum whose size is a multiple of 4 is generated into a 4-byte boundary alignment section, whose section name is the original section name with **\$4** postfixed. When the CPU type is H8SX, the speed of access to a 4-byte datum aligned on a 4-byte boundary address is improved.

A datum whose size is odd is generated into a 1-byte boundary alignment section, whose section name is the original section name with **\$1** postfixed. This can reduce the empty area.

The remaining data whose size is even and is not a multiple of 4 remains in the original section.

If the section name is changed by **#pragma section** or the **section** option, **\$4** or **\$1** will be appended to the changed section name.

When this option is not specified, **align** is assumed.

- Remarks

When the CPU type is not H8SX, **align=4** cannot be specified.

To locate the 1-byte or 4-byte data section at specific addresses with **align=4** specified, each section needs to be explicitly specified with the **start** option of the optimizing linkage editor.

In order to remain the boundary data construction unchanged, specify **noalign**.

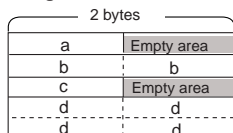
- Example

```
char a;
short b;
char c;
long d;
#pragma section _v
short e;
long f;
#pragma section

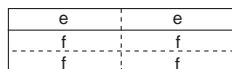
main()
{
    ...
}
```



- **noalign** specified

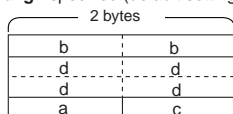


<Section B>  
Size: 10 bytes  
Boundary alignment: 2

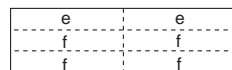


<Section B\_v>  
Size: 6 bytes  
Boundary alignment: 2

- **align** specified (default setting)

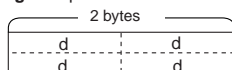


<Section B>  
Size: 8 bytes  
Boundary alignment: 2

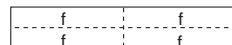


<Section B\_v>  
Size: 6 bytes  
Boundary alignment: 2

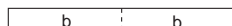
- **align=4** specified



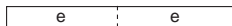
<Section B\$4>  
Size: 4 bytes  
Boundary alignment: 4



<Section B\_v\$4>  
Size: 4 bytes  
Boundary alignment: 4



<Section B>  
Size: 2 bytes  
Boundary alignment: 2



<Section B\_v>  
Size: 2 bytes  
Boundary alignment: 2



<Section B\$1>  
Size: 2 bytes  
Boundary alignment: 1

- o Data is located in the order of declaration.
- o 2-byte-aligned data is always located at an even address, thus generating an empty area being unused after odd-size data.

- o 2-byte aligned data is allocated before 1-byte aligned data in order to minimize the empty area.

- o Data are categorized into the following 3 groups:
  - X. data whose size is a multiple of 4
  - Y. data whose size is odd
  - Z. the others (data whose size is even but is not a multiple of 4)
- o The original data section is divided into the above 3 groups. For example, the B section will be divided into B\$4, B\$1 and B as shown below.
- X: The section consisting of data whose size is a multiple of 4 is aligned on a 4-byte boundary and "\$4" is appended after the original section name. (e.g. B\$4)
- Y: The section consisting of data whose size is odd is aligned on a 1-byte boundary and "\$1" is appended after the original section name. (e.g. B\$1)
- Z: The other data remains in the original section whose boundary alignment is 2-byte and the section name is unchanged. (e.g. B)



## LEgacy=v4: Code generation of Ver.4.0 Optimization technology

None

- Command Line Format

LEgacy=v4

- Description

If this option is specified along with 2600A, 2600N, 2000A, or 2000N as the CPU option, basic optimization processing is the same as in version 4 and earlier versions. When this option is not specified, the object code output by the compiler is subject to more optimization than with version 4.

- Remarks

This option is invalid when the CPU type is not 2600A, 2600N, 2000A, or 2000N.

When **legacy=v4** is specified, the following options become invalid:

opt\_range, del\_vacant\_loop, max\_unroll, infinite\_loop, global\_alloc, struct\_alloc, const\_var\_propagate, volatile\_loop, scope, noscope, strict\_ansi, file\_inline, file\_inline\_path, and enable\_register

## 2.2.3 List Options

**Table 2.4 List Options**

Item	Command Line Format	Dialog Menu	Specification
Listing file	<u>List</u> [= <file name>] NOList	C/C++ <List> [Generate list file]	Outputs a list file Not output a list file
Listing contents and format	SHow = <sub> [...] <sub>: { <u>S</u> ource   NOSource   <u>O</u> bject   <u>N</u> OObject   <u>S</u> tatistics   NOSTatistics   <u>A</u> llocation   <u>N</u> OAllocation    <u>E</u> xpansion   <u>N</u> OExpansion   Width = <numeric value>    Length = <numeric value>    Tab = { 4   <u>8</u> }  } }	C/C++ <List> [Contents :]	With/without source list With/without object list With/without statistics information With/without symbol allocation information With/without list after macro expansion Maximum characters per line: 0 or 80 to 132 Maximum lines per page: 0 or 20 to 255 Number of columns when using tabs: 4   8

### List, NOList: List File

C/C++ <List>[Generate list file]

- Command Line Format

List [= <list file name>]

NOList

- Description

Specifies whether a list file is output or not.

When **list** is specified, a list file name can be specified.

When **no list** is specified, a list file will not be output.

A list file name should be specified in accordance with section 8.1, Naming Files.

If no list file name is specified in **list**, a list file with the same name as the source file and a standard extension (lis/1st/lpp) is created. The standard extension for the UNIX version is “lis”, that for the PC version at C compilation is “1st”, and that for PC version at C++ compilation is “lpp”.

If this option is not specified, the compiler assumes **list** is specified.

## SHow: List Contents and Format

C/C++ <List> [Contents :]

- Command Line Format

```
SHow=    <sub> :    {    Source | NOSource |  
                Object      | NOObject |  
                Statistics  | NOSTatistics |  
                Allocation   | NOAllocation |  
                Expansion    | NOExpansion |  
                Width= <numeric value> |  
                Length= <numeric value> |  
                Tab= { 4 | g } }
```

- Description

Specifies the contents and format of the list output by the compiler, and the cancellation of list output.

For examples of each list in this section, refer to section 8.2, Compiler Listings.

If this option is not specified, the compiler assumes **show=source, noobject, statistics, noallocation, noexpansion, width=0, length=0, tab=8** are specified.

- Description

Table 2.5 shows a list of suboptions.

**Table 2.5 List of Suboptions of show Option**

Suboption	Description
source	Outputs a list of source programs
nosource	Does not output list of source programs
object	Outputs a list of object programs
noobject	Does not output list of object programs
statistics	Outputs a list of statistics information
nostatistics	Does not output list of statistics information
allocation	Outputs a list of symbol allocation information
noallocation	Does not output list of symbol allocation information
expansion	Outputs a source program list of include files and results of macro expansion. If the <b>nosource</b> suboption and the <b>expansion</b> suboption are specified simultaneously, the <b>expansion</b> suboption will be invalid, and no source program list will be output to a file.
noexpansion	Outputs a source program list before include files or macros have been expanded. If the <b>nosource</b> suboption and the <b>noexpansion</b> suboption are specified simultaneously, the <b>noexpansion</b> suboption will be invalid, and no source program list will be output to a file.
width=<numeric value>	The number specified by <numeric value> is set as the maximum number of characters in a single line of a list. The <numeric value> can specify decimal numbers from 80 to 132 or 0. If <numeric value> is specified as 0, the maximum number of characters in a single line is not limited.
length=<numeric value>	The number specified by <numeric value> is set as the maximum number of lines on a single page of a list. The <numeric value> can specify decimal numbers from 20 to 255 or 0. If <numeric value> is specified as 0, the maximum number of lines on a single page of a list is not limited.
Tab={4   8}	Specifies the tab size when displaying a list.

## 2.2.4 Optimize Options

**Table 2.6 Optimize Options**

Item	Command Line Format	Dialog Menu	Specification
Optimization	Optimize = { 0   1 }	C/C++ <Optimize> [Optimization]	Outputs object without optimization. Outputs object with optimization.
Inter-module optimization information	Goptimize	C/C++ <Optimize> [Generate file for inter-module optimization]	Outputs inter-module optimization supplementary information.
Optimization for speed	SPeed [= <sub>[, ...] ] <sub>: { Register    Shift    Loop [= { 1    2 }]    SWitch    Inline [=<numeric value>]    STruct    Expression }	C/C++ <Optimize> [Speed or size :] [Speed sub-options :]	Specifies code creation optimized for speed is specified. Performs register save and restore by push and pop expansion. Enhances the execution time of shift operation. Eliminates induction variables in a loop statement. Eliminates induction variables in a loop statement and expands the loop. Shortens the execution time of switch statement. Automatic inline expansion Shortens the execution time of structure assignment expression. Shortens the execution time of arithmetic operations, comparison, and assignment expressions.
switch statement output code selection	CAse = { <u>Auto</u>    Ifthen    Table }	C/C++ <Optimize> [Switch statement :]	Determined by whether or not <b>speed</b> is specified. Expanded with <b>if_then</b> comparisons. Expanded with jump table.
Memory indirect addressing mode	INDirect = { <u>Normal</u>   Extended }	C/C++ <Optimize> [Function call :]	Expands function call in memory indirect addressing mode. Expands function call in extended memory indirect addressing mode.

**Table 2.6 Optimize Options (cont)**

Item	Command Line Format	Dialog Menu	Specification
Pointer size	PTr16	C/C++ <Optimize> [2byte pointer]	Specifies the size of a pointer to data as two bytes.
Short absolute addressing mode	ABS8 ABS16	C/C++ <Optimize> [Data access :]	Accesses 8-bit data by the 8-bit absolute address. Accesses all data by the 16-bit absolute address.
External variable optimization	Volatile <u>NOVolatile</u>	C/C++ <Optimize> [Details...] [Global variables] [Treat global variables as volatile qualified]	Disables external variable optimization. Enables external variable optimization.
External variable optimization range	OPT_Range = { <u>All</u>   NOLoop    NOBlock }	C/C++ <Optimize> [Details...] [Global variables] [Specify optimizing range:]	Optimizes external variables within the entire function. Disables loop control variables or external variables in a loop from being moved outside the loop. Disables optimization of external variables which extend across loops or branches.
Vacant loop elimination	DEL_vacant_loop = { <u>0</u>   1 }	C/C++ <Optimize> [Details...] [Miscellaneous] [Delete vacant loop]	Disables elimination of vacant loops. Eliminates vacant loops.
Maximum number of loop expansions	MAX_unroll = <numeric value> <numeric value>: 1 to 32	C/C++ <Optimize> [Details...] [Miscellaneous] [Specify maximum unroll factor :]	Specifies the maximum number of times a loop is expanded. Default: 1 (2 when <b>speed</b> or <b>speed=loop[=2]</b> is specified)
Elimination of INFinite_loop = { <u>0</u> expression preceding infinite loop   1 }		C/C++ <Optimize> [Details...] [Global variables] [Delete assignment to global variables before an infinite loop]	Disables elimination of an assignment expression for external variables preceding an infinite loop. Eliminates an assignment expression for external variables preceding an infinite loop.

**Table 2.6 Optimize Options (cont)**

Item	Command Line Format	Dialog Menu	Specification
External variable register allocation	GLOBAL_Alloc = { 0   1 }	C/C++ <Optimize> [Details...] [Global variables] [Allocate registers to global variables]	Disables allocation of external variables to registers. Allocates external variables to registers.
Structure/union member register allocation	STRUCT_Alloc = { 0   1 }	C/C++ <Optimize> [Details...] [Global variables] [Allocate registers to struct/union members]	Disables allocation of structure/union members to registers. Allocates structure/union members to registers.
const variable constant propagation	CONST_Var_propagate = { 0   1 }	C/C++ <Optimize> [Details...] [Global variables] [propagate variables which are const qualified]	Disables constant propagation of external constants declared by <b>const</b> . Performs constant propagation of external constants declared by <b>const</b> .
Inline expansion of specific library functions	LIBrary = { <u>Function</u>   Intrinsic }	C/C++ <Optimize> [Details...] [Miscellaneous] [Inline memcpy/strcpy]	Makes function calls for <b>memcpy</b> and <b>strcpy</b> . Performs inline expansion for <b>memcpy</b> and <b>strcpy</b> .
Division of optimizing ranges	<u>SCOpe</u> NOScope	—	Optimizing ranges are divided. Optimizing ranges are not divided.
Inter-file inline expansion	FILE_inline = <file name>[,...]	C/C++ <Optimize> [Details...] [Inline] [inline file path]	Specifies a file for inter-file inline expansion.

## OPTimize: Optimization

C/C++ <Optimize>[Optimization]

- Command Line Format

Optimize = { 0 | 1 }

- Description

Specifies the level of compiler optimization for the object program.

When **optimize=0** is specified, the compiler does not optimize the object program.

When **optimize=1** is specified, the compiler optimizes the object program.

If this option is not specified, the compiler assumes **optimize=1** is specified.

- Remarks  
When **optimize=0** is specified, **speed=inline** or **loop** is invalid.

## Goptimize: Inter-Module Optimization Information

C/C++ <Optimize>[Generate file for inter-module optimization]

- Command Line Format  
Goptimize
- Description  
Outputs the supplement information for the inter-module optimization.  
For the file specified with this option, the inter-module optimization is performed at linkage.

## SPeed: Optimization for Speed

C/C++ <Optimize>[Speed or size :][Speed sub-options :]

- Command Line Format  
SPeed = <sub> [,...]  

<sub>:	{	Register	
		SHift	
		Loop [= { 1   <u>2</u> } ]	
		SWitch	
		Inline [= <numeric value>]	
		STruct	
		Expression }	
- Description  
Specifies optimization for speed for the object created by the compiler.  
When **300ha**, **300hn**, or **300** is selected for the CPU/operating mode, **speed=register** uses the PUSH and POP instructions to save and restore the contents of the registers at the entry and exit of a function, instead of using a run-time routine.  
The **speed=shift** option expands the shift operation to a code that does not use a run-time routine.  
The **speed=loop=1** option eliminates induction variables.  
The **speed=loop=2** option eliminates induction variables and performs loop expansion.  
The **speed=switch** option performs optimization for speed for code expansion of the **switch** statement.  
The **speed=inline** option performs inline expansion for small-size functions.



The **speed=inline=<numeric value>** modifies the maximum size of the target function for inline expansion. If CPU is H8SX or H8S(without legacy=v4 option), <numeric value> means the percentage of increase in program size allowed by inline expansion. For example, with **speed=inline=50**, inline expansion is performed up to 50% increase in program size, or up to 1.5 times larger.

If CPU is H8/300, H8/300H or H8S(with legacy=v4 option), <numeric value> is specified as the number of function nodes (total number of words consisting of variables and operators except for definitions). This means that functions smaller than the threshold shown by the <numeric value> are expanded. Here, the amount of program size increase depends on the size of the function to be expanded and the frequency of the calls of those functions. Hence the upper bound of the increase cannot be explicitly specified as can in H8SX or H8S(without legacy=v4 option).

If <numeric value> is omitted, 100 is assumed if the CPU type is H8SX or H8S, and 110 is assumed otherwise.

For details on the conditions of inline expansion, refer to the description on the in-line expansion of functions in section 10.2.1 (2), Extended Specifications Related to Functions.

The **speed=struct** option expands structure-type or double-type assignment to a code that does not use run-time routines.

The **speed=expression** option expands arithmetic operation, comparison, and assignment expressions to a code that does not use run-time routines (some expressions are excluded from this option).

If only **speed** is specified, optimization for speed is performed for **speed=register**, **shift**, **loop**, **switch**, **inline**, **struct**, and **expression**. If this option is not specified, the compiler optimizes for size instead of speed.

- Remarks

When no optimization (**optimize=0**) is specified, **speed=loop** or **inline** is invalid.

## CAsE: Switch Statement Output Code Selection Method

C/C++ <Optimize>[Switch statement :]

- Command Line Format

CAsE = { Auto | Ifthen | Table }

- Description

Specifies a switch-statement-output code-selection method.

When **case=auto** is specified, the compiler automatically selects an optimization method to reduce the size of the object code.

If **speed** or **speed=switch** is specified, the compiler automatically selects optimization for speed.

When **case=ifthen** is specified, **switch** statement codes are created using the **if\_then** method, which repeats, for all **case** labels, comparing the evaluated value of the expression in the **switch** statement with the **case** label value and jumps to the statement of the **case** label if they match. This method increases the object code size depending on the number of **case** labels in the **switch** statement.

When **case=table** is specified, **switch** statement codes are created using the table method, which stores the **case** label jump destinations in a jump table and enables a jump to the statement of the **case** label that matches the expression in the **switch** statement by accessing the jump table only once. This method increases the jump table size in the constant area depending on the range of **case** labels in the **switch** statement, but the execution speed is always the same.

If this option is not specified, the compiler assumes **case=auto** is specified.

- Example

```
int a, b;
:
switch(a){
    case 1:    b=3; break;
    case 2:    b=2; break;
    case 3:    b=1; break;
}
```

The following shows an example of a code expansion of a source program (when **cpu=2600n**)

When **case=ifthen** is specified

```
MOV.W  @_a:16,R0
MOV.B  R0H,R0H
BNE     Ld
CMP.B   #1,R0L
BEQ     L1
CMP.B   #2,R0L
BEQ     L2
CMP.B   #3,R0L
BNE     L4
BRA     L3
L1:
```

When **case=table** is specified

```
MOV.W  @_a:16,R0
SUB.W   #H'1,R0
CMP.W   #H'2,R0
BHI     Ld
MOV.B   @(L1:16,ER0),R0L
EXTU.W  R0
ADD.W   #LWORD Lp,R0
JMP     @ER0
```

```
Lp:
:
L1: (jump table)
```

**Table 2.7 Comparison of switch Statement Expansion by Expression Value**

Value of a	if_then Method		table Method	
	Object File Size	Execution Cycle	Object File Size	Execution Cycle
1	22 bytes	9	29 (26 + 3) bytes	17
3		17		

## INDirect: Memory Indirect Addressing Mode

C/C++ <Optimize>[Function call :]

- Command Line Format

INDirect = { Normal | Extended }

- Description

Specifies the memory indirect addressing mode for calling functions from the source program.

If **indirect=normal** is specified, all functions are called in memory indirect addressing mode (@aa:8).

If **indirect=extended** is specified, all functions are called in extended memory indirect addressing mode (@@vec:7).

The compiler outputs an address table for memory-indirect calls of the functions defined in the source program in the sections below:

— If **indirect=normal** is specified, section “\$INDIRECT”

— If **indirect=extended** is specified, section “\$EXINDIRECT”

For details on how to specify the section name, refer to the description on the section switching in section 10.2.1 (1), Extended Specifications Related to Memory Allocation.

- Remarks

The address table can be stored in the address ranges below:

— Section “\$INDIRECT”: Area from 0x0000 to 0x00FF

— Section “\$EXINDIRECT”: Area from 0x000 to 0x01FF in the normal mode  
: Area from 0x200 to 0x03FF in the other modes

At linkage, explicitly specify the location of these sections in the relevant address range with the **start** option.

The **indirect=extended** specification is valid only when the CPU type is H8SX.

To specify memory indirect addressing mode for a specific function, use **#pragma indirect**, **\_indirect**, or **\_indirect\_ex**. These specifications are given priority compared to this option. For details, refer to section 10.2.1 (2), Extension Functions Related to Functions.

Use either **normal** or **extended** exclusively between the definition and the call of the same function.

## PTr16: Pointer Size Specification

C/C++ <Optimize>[2byte pointer]

- Command Line Format

PTr16

- Description

Sets the size of the pointer indicating data to two bytes.

- Remarks

If this option is not specified, the size of the pointer indicating data is four bytes. If this option is specified, the data section to be referenced must be explicitly located in the 16-bit absolute address area. Addresses where to locate sections are specified with the **start** option of the optimizing linkage editor. For details on the **start** option, refer to section 4.2.5, Section Options. For details on the 16-bit absolute address area, refer to section 19.3, Access Range of Short Absolute Addresses.

This option is valid only if the CPU/operating mode is H8SXA, H8SXX, H8S/2600A, or H8S/2000A.

Take care the use of the **pTr16** option so that the handling of the same data and caller-callee relationship of the same function are consistent among files because changing the size of the pointer-to-data from 4 to 2 affects not only the resource allocation, but also the method to pass a function parameter and the function return value.

## ABS8, ABS16: Short Absolute Addressing Mode

C/C++ <Optimize>[Data access :]

- Command Line Format

ABS8

ABS16

- Description

Accesses the data to be allocated to the static area in short absolute addressing mode.

When **abs8** is specified, the compiler generates codes in 8-bit absolute addressing mode (**@aa:8**) for accessing **char**, **unsigned char**, and composite data, which is 1-byte aligned, consisting of **char** or **unsigned char** elements or members.

When **abs16** is specified, the compiler generates codes for accessing data in 16-bit absolute addressing mode (**@aa:16**) for the CPU/operating mode of **H8SXA**, **H8SXX**, **2600a**, **2000a**, and **300ha**. For the CPU/operating mode of **H8SXN**, **H8SXM**, **2600n**, **2000n**, **300hn**, and **300**, **abs16** is invalid.

The data to be accessed in 8-bit absolute addressing (**abs8** option) is output to section name “\$ABS8C”, “\$ABS8D”, or “\$ABS8B”. The data to be accessed in 16-bit absolute addressing mode (**abs16** option) is output to section name “\$ABS16C”, “\$ABS16D”, or “\$ABS16B”.

The variables to be accessed in short absolute addressing mode can also be specified by **#pragma abs8** and **#pragma abs16**, and keywords of **\_\_abs8** and **\_\_abs16**. If both an option and #pragma/keyword are specified, the #pragma/keyword specification is given priority over the option.

- Remarks

The section output by this option must be allocated to the short absolute address area at linkage. For the range of the short absolute address area, refer to section 19.3, Access Range of Short Absolute Addresses. For section name specifications for the short absolute address area, refer to the description on section switching in section 10.2.1 (1), Extended Specifications Related to Memory Allocation.

## **Volatile, NOVolatile: External Variable Optimization**

C/C++ <Optimize>[ Details...][Global variables][Treat global variables as volatile qualified]

- Command Line Format

Volatile

NOVolatile

- Description

When **volatile** is specified, the compiler does not optimize external variables.

When **novolatile** is specified, the compiler optimizes external variables that do not have a volatile specifier.

When this option is not specified, the compiler assumes that **novolatile** is specified.

- Example

Source program

```
volatile int a;
int b;
void main(void){
    a;
    b;
}
```

When **volatile** is specified

```
mov.w @_a,R0
mov.w @_b,R0          ; b is accessed as a volatile variable
rts
```

When **novolatile** is specified

```
mov.w @_a,R0
rts                  ; As a result of optimization, the access to b may be deleted
```

## OPT\_Range: External Variable Optimization Range Specification

C/C++ <Optimize> [Details...][Global variables][Specify optimizing range :]

- Command Line Format

OPT\_Range = { All | NOLoop | NOBlock }

- Description

When **opt\_range=all** is specified, the compiler optimizes external variables within the entire function.

When **opt\_range=noloop** is specified, external variables in a loop and external variables used in a loop iteration condition are not to be optimized.

When **opt\_range=noblock** is specified, external variables extending across branches (including loops) are not to be optimized.

When this option is omitted, **opt\_range=all** is assumed.

- Examples

(1) Optimization extending across a branch (done when **opt\_range=all** or **opt\_range=noloop** is specified)

```
int A,B,C;
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = A;
}
```

<Source program image after optimization>

```
int A,B,C;
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = 1;    /* Reference of A is eliminated and A = 1 is propagated */
}
```

(2) Optimization in a loop (done when **opt\_range=all** is specified)

```
int A,B,C[100]; /* External variables */
void f( ) {
    int i;
    for (i=0;i<A;i++) {
        C[i] = B;
    }
}
```

<Source program image after optimization>

```
void f( ) {
    int i;
    int temp_A, temp_B; /* Local variables */
    temp_A = A; /* Reference of A by loop iteration condition is moved outside the loop */
    temp_B = B; /* Reference of B in the loop is moved outside the loop */
    for (i=0;i<A;i++) { /* Reference of A in the loop is eliminated */
        C[i] = temp_B; /* Reference of B in the loop is eliminated */
    }
}
```

- Remarks

This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

When **opt\_range=noloop** is specified, **max\_unroll=1** is always the default.

When **opt\_range=noblock** is specified, **max\_unroll=1**, **const\_var\_propagate=0**, and **global\_alloc=0** are always the default.

## DEL\_vacant\_loop: Vacant Loop Elimination

C/C++ <Optimize>[Details...][Miscellaneous][Delete vacant loop]

- Command Line Format  
DEL\_vacant\_loop = { 0 | 1 }
- Description  
When **del\_vacant\_loop=0** is specified, even when there is no statements inside the loop, a loop is not eliminated.  
When **del\_vacant\_loop=1** is specified, loops without statements inside are eliminated.  
When this option is omitted, **del\_vacant\_loop=0** is assumed.
- Remarks  
This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

## MAX\_unroll: Loop Expansion Maximum Number Specification

C/C++ <Optimize>[Details...][Miscellaneous][Specify maximum unroll factor :]

- Command Line Format  
MAX\_unroll = <numeric value>
- Description  
Specifies the maximum number of loop expansions. An integer from 1 to 32 can be specified for <numeric value>. If any other value is specified, an error will occur.  
When **del\_vacant\_loop=1** is specified, loops with no internal processing are eliminated.  
When this option is omitted, **max\_unroll=2** is assumed with **speed** or **speed=loop[=2]** specified. For any other cases, **max\_unroll=1** is assumed.
- Remarks  
This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).  
When **opt\_range=noloop** or **opt\_range=noblock** is specified, **max\_unroll=1** is always the default.



## INFinite\_loop: Elimination of Expression Preceding Infinite Loop

C/C++ <Optimize>[Details...][Global variables][Delete assignment to global variables before an infinite loop]

- Command Line Format

INFinite\_loop = { 0 | 1 }

- Description

When **infinite\_loop=0** is specified, an assignment expression for external variables that is located immediately before an infinite loop is not eliminated.

When **infinite\_loop=1** is specified, an assignment expression that is located immediately before an infinite loop and that is an assignment to the external variable that is not used in the infinite loop is eliminated.

When this option is omitted, **infinite\_loop=0** is assumed.

- Example

```
int A;
void f()
{
    A = 1;          /* Assignment expression to external variable A */
    while(1) {}     /* A is not referenced */
}
```

<Source program image when **infinite\_loop=1** is specified>

```
void f()
{
    /* Assignment expression to external variable A is eliminated */
    while(1) {}
}
```

- Remarks

This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

## GLOBAL\_Alloc: External Variable Register Allocation

C/C++ <Optimize>[Details...][Global variables][Allocate registers to global variables]

- Command Line Format  
GLOBAL\_Alloc = { 0 | 1 }
- Description  
When **global\_alloc=0** is specified, allocation of external variables to registers is disabled.  
When **global\_alloc=1** is specified, external variables are allocated to registers.  
When this option is omitted, **global\_alloc=1** is assumed.
- Remarks  
This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).  
When **opt\_range=noblock** is specified, **global\_alloc=0** is the default.

## STRUCT\_Alloc: Structure/Union Member Register Allocation

C/C++ <Optimize>[Details...][Global variables][Allocate registers to struct/union members]

- Command Line Format  
STRUCT\_Alloc = { 0 | 1 }
- Description  
When **struct\_alloc=0** is specified, allocation of structure or union members to registers is disabled.  
When **struct\_alloc=1** is specified, structure or union members are allocated to registers.  
When this option is omitted, **struct\_alloc=1** is assumed.
- Remarks  
This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).  
If **struct\_alloc=1** is specified and if **opt\_range=noblock** or **global\_alloc=0** is specified, only local structure or union members are allocated to registers.

## CONST\_Var\_propagate: const Constant Propagation

C/C++ <Optimize>[Details...][Global variables][Propagate variables which are const qualified]

- Command Line Format

CONST\_Var\_propagate = { 0 | 1 }

- Description

When **const\_var\_propagate=0** is specified, constant propagation for external variables declared by **const** is disabled.

When **const\_var\_propagate=1** is specified, constant propagation is performed even for external variables declared by **const**.

When this option is omitted, **const\_var\_propagate=1** is assumed.

- Example

```
const int x = 1;
int A;
void f() {
    A = x;
}
```

<Source program image when **const\_var\_propagate=1** is specified>

```
void f() {
    A = 1;          /* x = 1 is propagated */
}
```

- Remarks

This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

When **opt\_range=noblock** is specified, **const\_var\_propagate=0** is the default.

Variables declared by **const** in a C++ program cannot be controlled by this option (constant propagation is always performed).

## LIBrary: Specific Library Function Inline Expansion

C/C++ <Optimize>[Details...][Miscellaneous][Inline memcpy/strcpy]

- Command Line Format

LIBrary = { Function | Intrinsic }

- Description

Regarding library functions memcpy and strcpy:

— When **library=function** is specified, these functions are called as functions.

— When **library=intrinsic** is specified, inline expansion is performed for these functions.

- Remarks

Specifying **library=intrinsic** is valid only when the CPU type is H8SX.

## SCope, NOScope: Division of Optimizing Ranges

None

- Command Line Format

SCope

NOScope

- Description

When the **scope** option is specified, the compiler divides the optimizing ranges of the large-size functions into some blocks.

When the **noscope** option is specified, the compiler does not divide the optimizing ranges.

When the optimizing range is expanded, the object performance is generally improved although the compilation time becomes longer. However, if registers are insufficient, the object performance may not be improved.

Use this option at performance tuning because it affects the object performance depending on the program.

- Remarks

This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

## FILE\_inline: Inter-file Inline Expansion

C/C++ <Optimize>[Details...][Inline][Inline file path]

- Command Line Format

FILE\_inline=<file name>[,...]

- Description

Performs inline expansion for functions that extend across files for the files specified with <file name>.

- Remarks

When the **file\_inline** option is specified, inline expansion is only applied to the functions specified with **#pragma inline** or keyword **inline** included in the file specified by <file name>. If the **-speed=inline** option is specified simultaneously, inline expansion is applied to all possible functions in the file.

If a global function is defined twice or more in files as the <file name> sub-option, no operation is guaranteed (using a single function definition randomly selected for inline expansion).

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

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

A wild card (\* or ?) cannot be specified for <file name>.

If a file has **#pragma asm-endsasm**, **#pragma inline\_asm** or **\_\_asm**, it will not be expanded.

This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

## 2.2.5 Other Options

**Table 2.8 Other Options**

Item	Command Line Format	Dialog Menu	Specification
Comment nesting	COMment	C/C++ <Other> [Miscellaneous options :] [Allow comment nest]	Allows comment (/* */) nesting.
Embedded C++ language	ECpp	C/C++ <Other> [Miscellaneous options :] [Check against EC++ language specification]	Checks the syntax according to the EC++ language specifications and determines the used memory management libraries.
MAC register	MAcsave	C/C++ <Other> [Miscellaneous options :] [Interrupt handler saves/restores MACH and MACL registers if used]	Always keeps the <b>MAC</b> register contents unchanged after an interrupt function is called.
Disable of loop iteration condition optimization	VOLATILE_Loop	C/C++ <Other> [Miscellaneous options :] [Treats loop condition as volatile qualified]	Disables optimization of loop iteration condition.
Enumeration data size	Byteenum	C/C++ <Other> [Miscellaneous options :] [Treat enum as char if it is in the range of char]	Handles enumeration data declared by enum with char.
Increase of registers for register variables	<u>Regexpansion</u> NORegexpansion	C/C++ <Other> [Miscellaneous options :] [Increase a register for register variable]	Uses (E)R3 to (E)R6 Uses (E)R4 to (E)R6
Common subexpression elimination	CMncode	C/C++ <Other> [Miscellaneous options :] [Put common subexpression on a register temporarily]	Optimizes with common subexpression elimination.

**Table 2.8 Other Options (cont)**

Item	Command Line Format	Dialog Menu	Specification
Block transfer instruction	EEpmov	C/C++ <Other> [Miscellaneous options :] [Use EEPMOVE in block copy]	Expands structure assignment expression by the eepmov instruction.
Restriction for output at preprocessor expansion	NOLINE	C/C++ <Other> [Miscellaneous options :] [Suppress #line in preprocessed source file]	Disables <b>#line</b> output at preprocessor expansion.
Message level	CHAnge_message = <sub>[,...] <sub>:<level> [=<n>[-m],...] <level>:{Information   Warning   Error }	C/C++ <Other> [User defined options :]	Changes message level.
Preferential allocation of register storage class variables	ENABle_register	C/C++ <Other> [Miscellaneous options :] [Enable register declaration]	Preferentially allocates the variables with register storage class specification to registers.
ANSI conformance	STRlct_ansi	C/C++ <Other> [Miscellaneous options :] [Obey ansi specifications more strictly]	Conforms to the ANSI standard for the following processing: <ul style="list-style-type: none"> <li>• Associative rule of floating-point operations</li> </ul>

## COMment: Comment Nesting

C/C++ <Other>[Miscellaneous options :] [Allow comment nest]

- Command Line Format

COMment

- Description

Enables nested comments to be written.

When this option is omitted, if nested comments are written, an error will occur.

- Example

```
/* This is an example of/* nested */ comment */
```

↑

(1)

When **comment** is specified, the compiler handles the above line as a nested comment, however, when the option is not specified, the compiler assumes (1) as the end of the comment.

## ECpp: Embedded C++ Language

C/C++ <Other>[Miscellaneous options :] [Check against EC++ language specification]

- Command Line Format  
ECpp
- Description  
Checks the syntax of the C++ source program according to the Embedded C++ language specifications. The Embedded C++ language specifications do not support **catch**, **const\_cast**, **dynamic\_cast**, **explicit**, **mutable**, **namespace**, **reinterpret\_cast**, **static\_cast**, **template**, **throw**, **try**, **typeid**, **typename**, and **using**. Therefore, if these keywords are written in the source program, the compiler will output an error message.  
This option also determines the memory management libraries used in EC++/C++ programs. This option must be specified to use an EC++ library.
- Remarks  
The Embedded C++ language specifications do not support a multiple inheritance or virtual base class.  
If a multiple inheritance or virtual base class is written in the source program, the compiler will show the error message "C5882 (E) Embedded C++ does not support multiple or virtual inheritance" at compilation.  
This option and the **exception** option cannot be specified simultaneously.

## MAcsave: MAC Register

C/C++ <Other>[Miscellaneous options :]

[Interrupt handler saves/restores MACH and MACL registers if used]

- Command Line Format  
MAcsave
- Description  
The contents of the **MAC** register always remain unchanged after an interrupt function is called.  
When **macsave** is specified, and if the **MAC** register is used in an interrupt function or if a function is called in the interrupt function, a save and restore code is created for the **MAC** register.  
If **macsave** is not specified, a save and restore code is created for the **MAC** register only when the **MAC** register is used in an interrupt function.



## **VOLATILE\_Loop: Disabling Optimization against Loop Iteration Condition**

C/C++ <Other>[Miscellaneous options :][Treats loop condition as volatile qualified]

- Command Line Format

VOLATILE\_Loop

- Description

Disables optimization of the loop iteration condition if it includes an external variable.

Note however that if type conversion is performed, if two or more external variables are included, or if composite operation is performed, optimization may be performed.

- Remarks

This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

If this option is specified, external variables within the loop are not optimized even though the **volatile** option has not been specified.

Without this option, if the loop iteration condition is invariant in the loop, the loop iteration condition may be eliminated.

## **Byteenum: Enumeration Data Size**

C/C++ <Other>[Miscellaneous options :] [Treat enum as char if it is in the range of char]

- Command Line Format

Byteenum

- Description

Handles the declared **enum** data as **char** data or **unsigned char** data.

If this option is specified, the compiler selects the **enum** data type according to the range of the members of the **enum** data. If the value is in the range from -128 to 127, the compiler handles the data as **char** data, whereas if the value is in the range from 0 to 255, the data is handled as **unsigned char** data.

When this option is not specified or at least one of the enum data members exceeds the above range, even if this option is specified, the **enum** data is handled as **int** data.

- Example

Source program

```
enum EM {a,b,c} E;
void main(void){E=b;}
```

When **byteenum** is specified

```
mov.b #1,R0L ; Transfers a 1-byte data
mov.b R0L,@_E
rts
_E:
.res.b 1 ; Allocates a 1-byte area to E
```

When **byteenum** is not specified

```
mov.w #1,R0 ; Transfers a 2-byte data
mov.w R0,@_E
rts
_E:
.res.w 1 ; Allocates a 2-byte area to E
```

## Regexpansion, NORegexpansion: Increasing Number of Registers for Register Variables

C/C++ <Other>[Miscellaneous options :] [Increase a register for register variable]

- Command Line Format

Regexpansion

NORegexpansion

- Description

When **regexpansion** is specified, the compiler increases the number of registers to which register variables are allocated.

When **noregexpansion** is specified, the compiler does not increase the number of registers to which register variables are allocated.

Generally, variable-access speed increases when the number of registers is increased.

For details on register variable allocation, refer to section 9.3.2 (3), Rules concerning registers.

When this option is not specified, the compiler assumes that **regexpansion** is specified.

- Remarks

The **regexpansion/noregexpansion** specification is invalid when the CPU type is H8SX or H8S.

## CMncode: Common Expression Optimization

C/C++ <Other>[Miscellaneous options :] [Put common subexpression on a register temporarily]

- Command Line Format

CMncode

- Description

Increases the number of target expressions for the optimization that converts a common subexpression into a temporary variable.

In general, when the number of target expressions for common subexpression optimization is increased by specifying this option, the temporary variables are allocated to registers and the performance of the object program is improved. However, when there are not enough registers, temporary variables are allocated to memory and the performance may be lowered. Use this option examining the performance of the program at performance tuning.

- Remarks

This option is valid only when the CPU type is H8/300, H8/300H or CPU type is H8S (with legacy=v4 option)).

## EEpmov: Block Transfer Instruction

C/C++ <Other>[Miscellaneous options :] [Use EEPMOVE in block copy]

- Command Line Format

EEpmov

- Description

Expands the assignment statements of structures and initial value assignment expressions for the arrays declared by local variables as the block transfer instruction(s). If the CPU is H8SX, the **MOVMD** instruction is used. Otherwise, the **EEPMOV** instruction is used. If the transfer size is too large for a block transfer instruction, a run-time routine will be used.

When this option is not specified, the compiler expands then to the **MOV** instructions or run-time routines.

- Remarks

For H8/300H and H8S(with legacy=v4 option), if an interrupt is accepted during the **EEPMOV.W** instruction, the control moves to the next instruction after returning from the interrupt, and therefore the EEPMOV operation result cannot be guaranteed. For source files including the functions which may accept an interrupt, this option should not be specified. For H8SX and H8S(without legacy=v4 option), expanded code can work if an interrupt occurs.

## NOLINE: Restriction for Output at Preprocessor Expansion

C/C++ <Other>[Miscellaneous options :] [Suppress #line in preprocessed source file]

- Command Line Format  
NOLINE
- Description  
When this option is specified, disables **#line** output at preprocessor expansion.
- Remarks  
This option is valid only when **preprocessor** is specified.

## CHAnge\_message: Message Level

C/C++ <Other>[Use defined options :]

- Command Line Format  
CHAnge\_message = <sub>[,...]  
    <sub> : <error level>[=<error number>[- <error number>]][,...]]  
    <error level> : { Information | Warning | Error }
- Description  
Changes the message level of information-level and warning-level messages.
- Example  
change\_message=information=1001,5038-5047  
Warning-level messages with the specified error numbers C1001 and from C5038 to C5047 are changed to **information**-level messages.  
change\_message=warning=5007-5009  
Information-level messages with the specified error numbers from C5007 to C5009 are changed to **warning**-level messages.  
change\_message=error=2-1024  
Information-level and warning-level messages with the specified error numbers from C0002 to C1024 are changed to **error**-level messages.  
change\_message=information  
All the warning-level messages are changed to **information**-level messages.  
change\_message=warning  
All the information-level messages are changed to **warning**-level messages.  
change\_message=error  
All the information-level and warning-level messages are changed to **error**-level messages.

- **Remarks**  
Output of the messages which were changed to the information-level can be suppressed by the **nomessage** option.  
An error number which is not defined is ignored.  
When this option is specified more than once, all the specifications are valid. If a number is specified more than once, the last specification is valid.

### **ENable\_register: Preferential Allocation of register Storage Class Variables**

C/C++ <Other> [Miscellaneous options :][Enable register declaration]

- **Format**  
ENable\_register
- **Description**  
Preferentially allocates the variables with register storage class specification to registers.
- **Remarks**  
If a variable cannot be allocated to a register, message C0101 (I) Register is not allocated to "variable name" in "function name" will be output. Note, however, that this message will not be output if a parameter is not allocated to a register. This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

### **STRICT\_ansi: ANSI Conformance**

C/C++ <Other> [Miscellaneous options :][Obey ansi specifications more strictly]

- **Format**  
STRICT\_ansi
- **Description**  
Conforms to the ANSI standard for the following processing:  
— Associative rule of floating-point operations
- **Remarks**  
When this option is specified, the operation results may be different from Ver.6.0 compiler or earlier.  
This option is valid only when the CPU type is H8SX or H8S(without legacy=v4 option).

Table 2.9 CPU Options

Item	Command Line Format	Dialog Menu	Specification
CPU/operating mode	CPu = { AE5   H8SXN[:<*2>]   H8SXM[:<*1>][:<*2>]   H8SXA[:<*1>][:<*2>]   H8SXX[:<*1>][:<*2>]   2600N   2600A[:<*1>]   2000N   2000A[:<*1>]   300HN-20   300HA[:<*1>]   300   300L   300Reg }	CPU [CPU:] [Multiple/Divide :]	AE5*3 H8SX normal mode H8SX middle mode H8SX advanced mode H8SX maximum mode H8S/2600 normal mode H8S/2600 advanced mode H8S/2000 normal mode H8S/2000 advanced mode H8/300H normal mode H8/300H advanced mode H8/300
Parameter storage register	REGParam = { 2   3 }	CPU [Change number of parameter registers from 2 (default) to 3]	Uses (E)R0 and (E)R1 2Uses (E)R0, (E)R1, and (E)R2
Allocating structure parameter or return value to register	STRUctreg <u>NOSTRUctreg</u>	CPU [Pass struct parameter via register]	Allocates 4-byte or less structure parameter or return value to register.
Allocating 4-byte parameter or return value to register	LONGreg <u>NOLONGreg</u>	CPU [Pass 4-byte parameter/ return value via register]	Allocates 4-byte parameter or return value to register (cpu=300).
double to float conversion	DOuble=Float	CPU [Treat double as float]	Handles double-type variable as float-type variable.
Stack size specification	STAck = { Small   <u>Medium</u>   Large }	CPU [Stack calculation]	Specifies stack calculation size: 1 byte 2 bytes 4 bytes

Notes: 1. Bit width of address space  
 2. Specification of multiplier and divider  
 3. For details on AE5, refer to section 17, Supporting AE5 Features.

**Table 2.9 CPU Options (cont)**

Item	Command Line Format	Dialog Menu	Specification
Runtime type information	RTti = { ON   <u>OFF</u> }	CPU [Enable/disable runtime information]	Enables dynamic_cast and typeid. Disables dynamic_cast and typeid.
Exception processing	EXception  <u>NOEXception</u>	CPU [Use try, throw and catch of C++]	Enables exception processing function Disables exception processing function.
Boundary alignment of structure, union, and class members	PAck = { 1   <u>2</u> }	CPU [Pack struct, union and class]	Assumes the boundary alignment value to be 1. Follows the boundary alignment.
8-bit absolute area address	SBr = <address>	CPU [Specify SBR address :]	Specifies the start address of the 8-bit absolute area.
Bit field order specification	Bl_t_order {= <u>Left</u>   Right >}	C/C++ <Object> [Bit field alloc-order :]	Stores members from upper bit. Stores members from lower bit.

## CPU: CPU/Operating Mode

CPU [CPU:][Multiple/Divide :]

- Command Line Format

CPu = {AE5  
H8SXN [: <multiplier and divider specification>] |  
H8SXM [: <address space bit width> ][: <multiplier and divider specification>] |  
H8SXA [: <address space bit width> ][: <multiplier and divider specification>] |  
H8SXX [: <address space bit width> ][: <multiplier and divider specification>] |  
2600N |  
2600A [: <address space bit width> ] |  
2000N |  
2000A [: <address space bit width> ] |  
300HN |  
300HA [: <address space bit width> ] |  
300 | 300L | 300Reg }  
    <address space bit width> : {20 | 24 | 28 | 32}  
    <multiplier and divider specification> : {M | D | MD} M:multiplier, D:divider

- Description

Specifies the CPU type and operating mode for the object program to be created.

If no input is made for the multiplier and divider specification, assumed there is no multiplier and divider.

Sub-options and their specifiable bit widths are listed in table 2.10.



**Table 2.10 Suboptions for cpu Option**

Suboption	Description	Bit Width	Multiplier/Divider
AE5	Object for AE5	—	—
H8SXN	H8SX normal mode	—	M, D, MD
H8SXM	H8SX middle mode	20, <u>24</u>	M, D, MD
H8SXA	H8SX advanced mode	20, <u>24</u> , 28, 32	M, D, MD
H8SXX	H8SX maximum mode	28, <u>32</u>	M, D, MD
2600n	H8S/2600 normal mode	—	—
2600a	H8S/2600 advanced mode	20, <u>24</u> , 28, 32	—
2000n	H8S/2000 normal mode	—	—
2000a	H8S/2000 advanced mode	20, <u>24</u> , 28, 32	—
300hn	H8/300H normal mode	—	—
300ha	H8/300H advanced mode	20, <u>24</u>	—
300	Object for H8/300	—	—
300l	Object for H8/300	—	—
	Provided to maintain the compatibility with the assembler.		
300reg	Object for H8/300	—	—
	Provided to maintain the compatibility with the older version of the C compiler.		

Note: When the bit width is not specified, the underlined default value is assumed.

- Example

```
-cpu = H8SXM:20      ; Without multiplier and divider, H8SX middle mode with 20-bit width
-cpu = h8sxa:32:md   ; With multiplier and divider, H8SX advanced mode with 32-bit width
-cpu = H8SXA:D       ; With divider, H8SX advanced mode with 24-bit width
```

- Remarks

When the **cpu** option is not specified, the compiler uses the H38CPU environment variable specifications. When the **cpu** option and the H38CPU environment variable are specified, the compiler uses the **cpu** specifications. When neither the **cpu** option nor the H38CPU environment variable is specified, an error will occur. For the CPU sub-option of AE-5, see section 17, Supporting AE5 Features.

## REGParam: Parameter Storage Register

CPU [Change number of parameter registers from 2(default) to 3]

- Command Line Format

```
REGParam = { 2 | 3 }
```

- Description

Specifies the number of registers for storing parameters.

If **regparam=2** is specified, parameters are passed in two registers: ER0 and ER1 (R0 and R1 for the H8/300).

If **regparam=3** is specified, parameters are passed in three registers: ER0, ER1, and ER2 (R0, R1, and R2 for the H8/300).

When this option is not specified, **regparam=2** is assumed.

## **STRUctreg, NOSTRUctreg: Register Allocation of Structure Parameters**

CPU [Pass struct parameter via register]

- Command Line Format

STRUctreg

NOSTRUctreg

- Description

Specifies whether structure parameters or return values are allocated to registers or not.

If **nostructreg** is specified, parameters are passed via a memory instead of a register.

If **structreg** is specified, parameters can be passed via a register.

The size of structures which can be passed as parameters are 2 bytes when CPU=300, and 4 bytes for other CPU specifications.

When this option is omitted, **nostructreg** is assumed.

- Remarks

If the CPU is H8/300 and the **longreg** is specified, up to 4 bytes of data can be allocated to a register as a parameter and a return value.

## **LONGreg, NOLONGreg: Register Allocation of 4-Byte Parameters**

CPU [Pass 4-byte parameter/return value via register]

- Command Line Format

LONGreg

NOLONGreg

- Description

Specifies whether 4-byte parameters or return values are allocated to registers or not.

The type of variable to be allocated to a register by this option is **long**, **unsigned long**, and **float**.

If **nolongreg** is specified, parameters are passed via a memory instead of a register.

If **longreg** is specified, parameters can be passed via a register.

When this option is omitted, **nolongreg** is assumed.

- Remarks

This option can be specified only when the CPU is H8/300.

When the CPU is not H8/300, 4-byte data can always be allocated to registers.

## **DOuble=Float: double to float Conversion**

CPU [Treat double as float]

- Command Line Format

DOuble=Float

- Description

Generates an object after converting double-type (double-precision floating-point) variables/values to float-type (single-precision floating-point) ones.

## STack: Stack Size Specification

CPU [Stack calculation :]

- Command Line Format  
STack = { Small | Medium | Large }

- Description

Specifies the stack size.

When **stack=small** is specified, stack addresses are calculated only in the least significant 1 byte without a carry to the upper bytes.

When **stack=medium** is specified, stack addresses are calculated only in the least significant 2 bytes without a carry to the upper bytes.

When **stack=large** is specified, stack addresses are calculated as 4byte value.

When this option is omitted, **stack=medium** is assumed.

- Remarks

This option should be specified to the whole program with the same suboption.

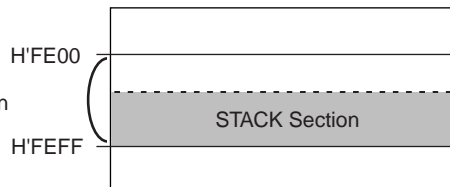
If stack address calculation is performed with a size larger than the specified size, or a variable is allocated beyond the 1-byte, 2-byte and 4-byte address boundary values, the compiler does not output an error or warning message. Note, however, that the **goptimize** option allows the output of these warning messages by the optimizing linkage editor.

In this case, increase the size of the stack.

Example:

-stack=small

Allocate the stack so that stack addresses can be operated within 1 byte.



## RTti: Runtime Type Information

CPU [Enable/disable runtime information]

- Command Line Format

RTti = { ON | Off }

- Description

Enables or disables runtime type information.

When **rtti=on** is specified, **dynamic\_cast** and **typeid** are enabled.

When **rtti=off** is specified, **dynamic\_cast** and **typeid** are disabled.

When this option is omitted, **rtti=off** is assumed.

- Remarks

Do not define object files which are created by specifying this option in a library, and do not output files with this information as relocatable object files. A symbol double definition error or symbol undefined error will occur.

## EXception, NOEXception: Exception Processing

CPU [Use try, throw and catch of C++]

- Command Line Format

EXception

NOEXception

- Description

When **noexception** is specified, the C++ exception processing functions are disabled.

When **exception** is specified, the C++ exception processing functions (**try**, **catch**, and **throw**) are enabled.

When an exception processing function is used, the code performance may be reduced.

When this option is omitted, **noexception** is assumed.

The **exception** option and **ecpp** option cannot be specified simultaneously.

## Pack: Boundary Alignment of Structure, Union, and Class Members

CPU [Pack struct, union and class]

- Command Line Format

Pack = { 1 | 2 }

- Description

Specifies the boundary alignment of structure, union, and class members.

Boundary alignment of structure members can also be specified by the **#pragma pack** extension. If both this option and **#pragma** are specified, only **#pragma** is valid.

The boundary alignment of structures, unions, and classes equals to the maximum boundary alignment of members.

For details, refer to section 10.1.2 (2), Compound Type (C), Class Type (C++).

When this option is not specified, the compiler assumes that **pack=2** is specified.

- Remarks

Table 2.11 shows the boundary alignment of structure, union, and class members when **pack** is specified.

**Table 2.11 Boundary Alignment of Structure, Union, and Class Members when the pack Option is Specified**

Member Type	pack=1	pack=2	Not Specified
[unsigned] char	1	1	1
[unsigned] short, [unsigned] int, [unsigned] long, floating-point type, pointer type	1	2	2
Structures, unions, and classes aligned to a 1-byte boundary	1	1	1
Structures, unions, and classes aligned to a 2-byte boundary	1	2	2

A member of a struct, union or class to which the pack=1 option or #pragma pack 1 is specified must not be accessed via a pointer (including an access via a pointer in a member function).

Example: (cpu=2600a and pack=1)

```
struct S {
    char x;
    int y;
} s;
int *p=&s.y; // the address of s.y can be an odd number
void test()
{
    s.y=1; // accessed correctly
    *p =1; // can be accessed incorrectly
}
```

## SBr: 8-Bit Absolute Area Address Specification

CPU [Specify SBR address :]

- Command Line Format

SBr = <address>

- Description

Specifies the start address of the 8-bit absolute area.

When **sbr=<address>** is specified, a 1-byte area starting from <address> is used as the 8-bit absolute area.

- Remarks

This option is valid only when the CPU type is H8SX.

An <address> should be within a data area.

When this option is omitted, the default 8-bit absolute address is assumed as <address>. For details on the 8-bit absolute address area, refer to section 19.3, Access Range of Short Absolute Addresses.

- Example

```
ch38 -sbr=A0000 test.c
```

Compiled assuming the 8-bit absolute address area begins at 0xA0000.

## Bit\_order: Bit Field Order Specification

CPU [Bit field alloc-order]

- Command Line Format  
Bit\_order = { Left | Right }
- Description  
Specifies the order of bit field members.  
When **bit\_order=left** is specified, members are allocated from the most significant bit.  
When **bit\_order=right** is specified, members are allocated from the least significant bit.  
When this option is not specified, the compiler assumes that **bit\_order=left** is specified.
- Remarks  
For details on allocation of bit field members, refer to section 10.1.2, Internal Data Representation, and the description on **#pragma bit\_order** in section 10.2.1, #pragma Extension Specifiers and Keywords.  
Keep the order of the same bit field members consistent among files.

### 2.2.7 Options Other Than Above

Table 2.12 Options Other Than Above

Item	Command Line Format	Dialog Menu	Specification
Selecting C or C++ language	LANG = { C   Cpp }	— (Determined by an extension)	Compiled as C source program. Compiled as C++ source program.
Disable of copyright output	<u>LOGO</u> NOLOGO	— (nologo is always valid)	Outputs copyright. Disables copyright output.
Character code select in string literal	EUc SJis LATin1	—	Selects euc code. Selects sjis code. Selects latin1 code.
Japanese character conversion within object code	OUtcode = { Euc   Sjis }	—	Selects euc code. Selects sjis code.
Subcommand file	SUBcommand = <file name>	—	Command option is fetched from the file specified with <file name>.



## LANG: Selecting C or C++ Language

None (Always determined by an extension)

- Command Line Format

LANG = { C | Cpp }

- Description

Specifies the language of the source program.

If **lang=c** is specified, the compiler will compile the program file as a C source program.

If **lang=cpp** is specified, the compiler will compile the program file as a C++ source program.

If this option is not specified, the compiler will determine whether the source program is a C or a C++ program by the extension of the source program file name. If the extension is c, the compiler will compile it as a C source program. If the extension is cpp, cc, or cp, the compiler will compile it as a C++ source program. If there is no extension, the compiler will compile the program as a C source program.

- Example
 

<code>ch38 test.c</code>	Compiled as a C source program.
<code>ch38 test.cpp</code>	Compiled as a C++ source program.
<code>ch38 -lang=cpp test.c</code>	Compiled as a C++ source program.
<code>ch38 test</code>	Assumed to be test.c and thus compiled as a C source program.
- Remarks
 

If **lang=c** is specified, **ecpp** is invalid.

## LOGO, NOLOGO: Disable of Copyright Output

None (nologo is always available)

- Command Line Format
 

LOGO  
NOLOGO
- Description
 

Disables the copyright output.

When **logo** is specified, copyright display is output.

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

When this option is omitted, **logo** is assumed.

## EUC, SJIS, LATIN1: Character Code Select in String Literal

None

- Command Line Format
 

EUC  
SJIS  
LATIN1
- Description
 

Use this option to specify the character code to be output to the object program for Japanese language or ISO-Latin1 code written in a string literal, a character constant, or a comment.

Table 2.13 shows character code in the string literal for three types of host computers.

**Table 2.13 Relationship between Host Computer and Character Code in String Literal**

Host Computer	Option Specification			
	euc	sjis	latin1	Not Specified
PC	euc	sjis	latin1	sjis
SPARC	euc	sjis	latin1	euc
HP9000/700	euc	sjis	latin1	sjis

- Remarks  
If **latin1** is specified, **outcode** will be invalid.

### OUTcode: Japanese Code Conversion in Object Code

None

- Command Line Format  
OUnode = Euc | Sjis
- Description  
Specifies the Japanese character code to be output to the object program when Japanese is written in string literal and character constants.  
If **outcode=euc** is specified, the compiler outputs the Japanese character code in the **euc** code.  
If **outcode=sjis** is specified, the compiler outputs the Japanese character code in the **sjis** code.  
**euc** or **sjis** can be specified for the Japanese character code in a source program.

### SUBcommand: Subcommand File

None

- Command Line Format  
SUBcommand = <subcommand file name>
- Description  
Specifies the subcommand file where options used at compiler initiation are stored. The command format in the subcommand file is the same as that on the command line.
- Example  
opt.sub:                                -show=object -debug -byteenum  
Command line specification: ch38 -cpu=2600a -subcommand=opt.sub test.c  
Interpretation by compiler: ch38 -cpu=2600a -show=object -debug  
                                     -byteenum test.c



## Section 3 Assembler Options

### 3.1 Command Line Format

The format of the command line to initiate the assembler is as follows:

```
asm38 [ $\Delta$ <option> ...] [ $\Delta$ <file name> [,...*]] [ $\Delta$ <option> ...]  
      <option>: -<option> [=<suboption> [,...]]
```

**Note\*:** When the user specifies multiple source files, the assembler will merge and assemble these files as one unit in the order they were specified. In this case, the user must write the .END assembly directive only in the file that was specified last.

### 3.2 List of Options

Table 3.1 shows assembler option formats, abbreviations, and defaults. In the command line format, uppercase letters indicate the abbreviations. Characters underlined indicate the default assumptions.

The format of the dialog menus that correspond to the HEW is as follows:

Tab name [Item]

Options are described in the order of tabs in the HEW option dialog box.

### 3.2.1 Source Options

**Table 3.1 Source Options**

Item	Command Line Format	Dialog Menu	Specification
Include file directory	Include = <path name>[,...]	Assembly <Source> [Show entries for :] [Include file directories]	Specifies include-file destination path name.
Replacement symbol definition	DEFine = <sub>[, ...] <sub>: <replacement symbol> = <string literal>	Assembly <Source> [Show entries for :] [Defines]	Defines replacement string literal.
Integer preprocessor variable definition	ASsignA = <sub>[, ...] <sub>: <variable name> = <integer constant>	Assembly <Source> [Show entries for :] [Preprocessor variables]	Defines integer preprocessor variable.
Character preprocessor variable definition	ASsignC = <sub>[, ...] <sub>: <variable name> = <string literal>	Assembly <Source> [Show entries for :] [Preprocessor variables]	Defines character preprocessor variable.

## Include

Assembly <Source> [Show entries for :] [Include file directories]

- Command Line Format  
Include = <path name> [...]
- Description

The **include** option specifies the include file directory. The directory name depends on the naming rule of the host machine used. As many directory names as can be input in one command line can be specified. The current directory is searched first, and then the directories specified by the **include** option are searched in the specified order.

Example:     asm38 aaa.mar -include=c:/usr/tmp,c:/tmp

(.INCLUDE "file.h" is specified in aaa.mar.)

The current directory, c:/usr/tmp, and c:/tmp are searched for file.h in that order.

### Relationship with Assembler Directives

Option	Assembler Directive	Result
include	(regardless of any specification)	(1) Directory specified by .INCLUDE
		(2) Directory specified by include*
(no specification)	.INCLUDE <file name>	Directory specified by .INCLUDE

Note: The directory string literals specified by the **include** option must come before the literal specified by **.INCLUDE** directive.

## DEFine

Assembly <Source> [Show entries for :] [Defines]

- Command Line Format

DEFine = <sub> [,...]

<sub>: <replacement symbol> = <string literal>

- Description

The **define** option defines the specified symbol as the corresponding string literal to be replaced by the preprocessor.

Differences between the **define** option and the **assignc** option are the same as those between **.DEFINE** and **.ASSIGNC**.

### Relationship with Assembler Directives

Option	Assembler Directive	Result
define	.DEFINE directive*	String literal specified by define
	(no specification)	String literal specified by define
(no specification)	.DEFINE directive	String literal specified by .DEFINE

Note: When a string literal is assigned to a replacement symbol by the **define** option, the definition of the replacement symbol by **.DEFINE** is invalidated. This replacement is not applied to the **.AENDI**, **.AENDR**, **.AENDW**, **.AIFDEF**, **.END**, **.ENDM**, **.ENDI**, **.ENDS**, and **.ENDW**, directives.



## AAssignA

Assembly <Source> [Show entries for :][Preprocessor variables]

- Command Line Format

AAssignA = <sub>[,...]

<sub>: <preprocessor variable> = <integer constant>

- Description

The **assigna** option sets an integer constant to a preprocessor variable. The naming rule of preprocessor variables is the same as that of symbols. An integer constant is specified by combining the radix (B', Q', D', or H') and a value. If the radix is omitted, the value is assumed to be decimal. An integer constant must be within the range from -2,147,483,648 to 4,294,967,295. To specify a negative value, use a radix other than decimal.

Example:   asm38   aaa.mar   -assigna=\_\$=H'FF

Value H'FF is assigned to preprocessor variable \_\$ . All references (\&\_\$) to preprocessor variable \_\$ in the source program are set to H'FF.

- Remarks

If the host computer OS is UNIX, and if the dollar mark (\$) is in the preprocessor variable or the apostrophe (') of the radix is in the integer constant, a backslash (\) must be specified before the dollar mark (\$) or the apostrophe (') of the radix.

### Relationship with Assembler Directives

Option	Assembler Directive	Result
assigna	.ASSIGNA*	Integer constant specified by assigna
	(no specification)	Integer constant specified by assigna
(no specification)	.ASSIGNA	Integer constant specified by .ASSIGNA

Note: When a value is assigned to a preprocessor variable by the **assigna** option, the definition of the preprocessor variable by **.ASSIGNA** is invalidated.

## A`sign`C

Assembly <Source> [Show entries for :][Preprocessor variables]

- Command Line Format

A`sign`C = <sub>

<sub>: <preprocessor variable> = <string literal>

- Description

The **assignc** option sets a string literal to a preprocessor variable.

The naming rule of preprocessor variables is the same as that of symbols.

A string literal must be enclosed with double-quotation marks (").

Up to 255 characters can be specified for a string literal.

Example:     asm38 aaa.mar -assignc=\_\$=ON!OFF

String literal ON!OFF is assigned to preprocessor variable \_\$ . All references (\&\_ \$) to preprocessor variable \_\$ in the source program are set to ON!OFF.

- Remarks

To specify the following characters in a string literal when the host computer OS is UNIX, specify a backslash (\) before the characters. To specify a string literal before or after the following characters, enclose the string literal with double-quotation marks (").

— Exclamation mark (!)

— Double-quotation mark (")

— Dollar mark (\$)

— Single quotation mark (')

## Relationship with Assembler Directives

Option	Assembler Directive	Result
assignc	.ASSIGNC*	String literal specified by assignc
	(no specification)	String literal specified by assignc
(no specification)	.ASSIGNC	String literal specified by .ASSIGNC

Note: When a string literal is assigned to a preprocessor variable by the **assignc** option, the definition of the preprocessor variable by **.ASSIGNC** is invalidated.

### 3.2.2 Object Options

**Table 3.2 Object Options**

Item	Command Line Format	Dialog Menu	Specification
Debugging information	Debug <u>NODebug</u>	Assembly <Object> [Debug information :]	Outputs debug information. Not output debug information.
Pre-processor expansion result	EXPand [ = <output file name>]	Assembly <Object> [Generate assembly source file after preprocess]	Outputs preprocessor expansion result.
Optimization	OPTimize <u>NOOPTimize</u>	Assembly <Object> [Optimize]	Optimized. Not optimized.
Displacement size setting	BR relative = <sub>  <sub>: { 8   16 }	Assembly <Object> [Default of branch displacement size :]	Sets the default size for the number of bits used to represent displacements for branch instructions. Set to 8 bits. Set to 16 bits.
Inter-module optimization	GOptimize	Assembly <Object> [Generate file for inter-module optimization]	Outputs additional information for inter-module optimization.
Object module output	<u>Object</u> [= <output file name>] NOObject	Assembly <Object> [Output file directory :]	Outputs an object file.  Not output an object file.

## Debug, NODebug

Assembly <Object> [Debug information :]

- Command Line Format

Debug  
NODebug

- Description

The **debug** option specifies output of debugging information. The **nodebug** option specifies no output of debugging information. The **debug** and **nodebug** options are only valid in cases where an object module is generated.

- Remarks

Debugging information is required when debugging a program with the debugger. Debugging information includes information about source statement lines and symbols.

### Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result
debug	(regardless of any specification)	Debugging information is output.
nodebug	(regardless of any specification)	Debugging information is not output.
(no specification)	.OUTPUT DBG	Debugging information is output.
	.OUTPUT NODBG	Debugging information is not output.
	(no specification)	Debugging information is not output.

## EXPand

Assembly <Object> [Generate assembly source file after preprocess]

- Command Line Format

EXPand [= <output file name>]

- Description

The **expand** option outputs an assembler source file for which macro expansion, conditional assembly, structured assembly, and file inclusion have been performed.

When this option is specified, no object will be generated.

When the output file parameter is omitted, the assembler takes the following actions:

— If the file extension is omitted:

The file extension will be exp.

— If the specification is completely omitted:

The source file name will be the same name as that of the input source file (the source file specified first) and the file extension will be exp.

- Remarks

Do not specify the same file name for the input and output files.

## OPTimize, NOOPTimize

Assembly <Object> [Optimize]

- Command Line Format

OPTimize

NOOPTimize

- Description

The **optimize** option specifies whether or not to optimize the PC relative format, displacement size of register-indirect with displacement, and address size of the absolute addressing format. Regarding the MOVA instruction of H8SX, the optimization is performed or not as shown in the table below.

The frist operand	Whether optimized or not
@(disp,Reg) *1	Yes
@(disp,@ERn.sz) *2	Yes
@(disp,@+ERn.sz) or @(disp,@-ERn.sz) *2	Yes
@(disp,@ERn+.sz) or @(disp,@ERn-.sz) *2	Yes
@(disp,@(disp,Reg).sz) *2 *3	No
@(disp,@abs.sz) *2	No

Note: 1. "Reg" can be RnL.B, RnH.B, Rn.W or En.W.  
2. "sz" can be either B or W.  
3. "Reg" can be ERn, RnL.B, Rn.W or ERn.L.

This option is valid for executable instructions when a displacement (:8 or :16) is not specified, or an allocated size (:8, :16, :24, or :32) of an absolute address is not specified. The displacement size is set as shown below according to the displacement value of the PC relative format.

When no optimization is specified in the H8S/2600 advanced mode:

Type of Displacement	Size
Absolute value (-32768 to 32767)	16 bits*
Relative value	16 bits
External reference value	16 bits

Note: Only valid when an absolute symbol that is defined after the instruction is referenced.

When optimization is specified in the H8S/2600 advanced mode:

Type of Displacement	Size
Absolute value	(-128 to 127)
	8 bits
	(-32768 to -129) (128 to 32767)
	16 bits
Relative value	16 bits
External reference value	16 bits

#### Example

```
asm38 aaa.mar -optimize
```

The object module is optimized.

```
asm38 aaa.mar
```

The object module is not optimized.

### Relationship with Assembler Directives

The assembler gives priority to specifications made by using options

Option 1	Option 2	Assembler Directive	Result
optimize	(regardless of any specification)	(regardless of any specification)	Optimized number of bits
nooptimize	br_relative	(regardless of any specification)	Number of bits specified by br_relative
	(no specification)	.DISPSIZE	Number of bits specified by .DISPSIZE
		(no specification)	8 bits

Note: The **optimize** option has priority over the **br\_relative** option for the output of the object module and the **.DISPSIZE** directive.

## BR\_relative

Assembly <Object> [Default of branch displacement size :]

- Command Line Format

BR\_relative = {8 | 16}

- Description

The **br\_relative** option specifies a default size for the displacements of the instructions that reference the symbol which is defined in advance.

— 8: The default size is 8 bits

— 16: The default size is 16 bits

This option is valid when a displacement size (:8 or :16) is specified and the **optimize** option has not been specified.

- Remarks

In the H8/300 and the H8/300L, **br\_relative** has a fixed value of 8, and thus has no meaning.

## Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option 1	Option 2	Assembler Directive	Result
optimize	(regardless of any specification)	(regardless of any specification)	Optimized number of bits
nooptimize	br_relative	(regardless of any specification)	Number of bits specified by br_relative
	(no specification)	.DISPSIZE	Number of bits specified by .DISPSIZE
		(no specification) cpu=300, 300L, 300HN, 2000N, 2600N, H8SXN	8 bits
		(no specification) cpu=300HA, 2000A, 2600A, H8SXM, H8SXA, H8SXX, AE5	16 bits

Note: The **optimize** option has priority over the **br\_relative** option for the output of the object module and the **.DISPSIZE** directive.



## GOptimize

Assembly <Object> [Generate file for inter-module optimization]

- Command Line Format

GOptimize

- Description

The **goptimize** option specifies outputs of additional information for the inter-module optimization. Inter-module optimization is performed when the files for which this option is specified are linked.

## Object, NOObject

Assembly <Object> [Output file directory :]

- Command Line Format

Object [= <output file name>]

NOObject

- Description

The **object** option specifies output of an object module.

The **noobject** option specifies no output of an object module.

When the object output file parameter is omitted, the assembler takes the following actions:

— If the file extension is omitted:

The file extension will be obj.

— If the specification is completely omitted:

The source file name will be the same name as that of the input source file (the source file specified first) and the file extension will be obj.

## Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result
object	(regardless of any specification)	An object module is output.
noobject	(regardless of any specification)	An object module is not output.
(no specification)	.OUTPUT OBJ	An object module is output.
	.OUTPUT NOOBJ	An object module is not output.
	(no specification)	An object module is output.

Note: Do not specify the same file name for the input source file and the output object module. If the same file is specified, the contents of the input source file will be lost.

### 3.2.3 List Options

**Table 3.3 List Options**

Item	Command Line Format	Dialog Menu	Specification
Assemble listing output control	LISt [= <output file name>] <u>NOLIS</u> t [ = <output file name>]	Assembly <List> [Generate list file]	Outputs a source program list. Not output a source program list.
Source program listing output control	<u>S</u> ource NOSource	Assembly <List> [Source program :]	Controls output of source program listing.
Part of source program listing output control*	<u>S</u> How [= <item>[, ...]] NOSHHow [= <item>[, ...]] <item>: {CONditionals   Definitions   CALLs   Expansions   Structured   CODE}	Assembly <List> [Source program list contents :] [Code]	Controls output of part of source program listing.
Cross-reference listing output control*	<u>C</u> Ross_reference NOCross_reference	Assembly <List> [Cross reference :]	Outputs a cross-reference listing. Not output a cross-reference listing.
Section information listing output control*	<u>S</u> ection NOSection	Assembly <List> [Section :]	Outputs a section information listing. Not output a section information listing.

Note: These options are valid only if the **list** option is specified.

## LISt, NOLISt

Assembly <List> [Generate list file]

- Command Line Format

LISt [= <output file name>]

NOLISt [= <output file name>]

- Description

The **list** option outputs an assemble listing.

The **nolist** option does not output an assemble listing. If the **nolist** option is specified and the specification is made to output the file name, the assemble listing is output to the file for only the line where the error occurred.

When the listing output file parameter is omitted, the assembler takes the following actions:

— If the file extension is omitted:

The file extension will be lis.

— If the specification is completely omitted:

The output file name will be the same name as that of the input source file (the source file specified first) and the file extension will be lis.

### Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result
list	(regardless of any specification)	An assemble listing is output.
nolist	(regardless of any specification)	An assemble listing is not output.
(no specification)	.PRINT LIST	An assemble listing is output.
	.PRINT NOLIST	An assemble listing is not output.
	(no specification)	An assemble listing is not output.

Note: Do not specify the same file for the input source file and the output object file. If the same file is specified, the contents of the input source file will be lost.

## S**Source**, N**OS**Source

Assembly <List> [Source program :]

- Command Line Format

S**Source**

N**OS**Source

- Description

The **source** option outputs a source program listing to the assemble listing.

The **nosource** option does not output a source program listing to the assemble listing.

The **source** and **nosource** options are only valid in cases where an assemble listing is being output.

## Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result (When an Assemble Listing Is Output)
source	(regardless of any specification)	A source program listing is output.
nosource	(regardless of any specification)	A source program listing is not output.
(no specification)	.PRINT SRC	A source program listing is output.
	.PRINT NOSRC	A source program listing is not output.
	(no specification)	A source program listing is output.

## SHow, NOSHow

Assembly <List> [Source program list contents :] [Code:]

- Command Line Format

**SHow** [= <output type>[,...]]

**NOSHow** [= <output type>[,...]]

<output type>: {CONditionals | Definitions | CALLs | Expansions | Structured | CODE}

- Description

Outputs or suppresses a part of preprocessor source statements in the source program listing, and outputs or suppresses a part of object code lines.

The items specified by <output type> will be output or suppressed depending on the option.

When no output type is specified, all items will be output or suppressed.

show:           Output

noshow:        No output (suppress)

The **show** option and **noshow** option is valid only if assemble listing is output. The following output types can be specified:

Output Type	Object	Description
conditionals	Unsatisfied condition	Unsatisfied .AIF or .AIFDEF statements
definitions	Definition	Macro definition parts, .AREPEAT and .AWHILE definition parts, .INCLUDE directive statements .ASSIGNA and .ASSIGNC directive statements
calls	Call	Macro call statements, .AIF, .AIFDEF, and .AENDI directive statements
expansions	Expansion	Macro expansion statements .AREPEAT and .AWHILE expansion statements
structured	Structured expansion	Structured assembly expansion statements
code	Object code lines	The object code lines exceeding the source statement lines

- Remarks

In a PC version, when specifying more than two output types, enclose the types with parentheses.

## Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result
show[=<output type>]	(regardless of any specification)	The object code is output.
noshow[=<output type>]	(regardless of any specification)	The object code is not output.
(no specification)	.LIST <output type> (output)	The object code is output.
	.LIST <output type> (suppress)	The object code is not output.
	(no specification)	The object code is output.

## CRoss\_reference, NOCRoss\_reference

Assembly <List >[Cross reference :]

- Command Line Format

CRoss\_reference

NOCross\_reference

- Description

The **cross\_reference** option specifies output of a cross-reference listing to the assemble listing.

The **nocross\_reference** option specifies no output of a cross-reference listing to the assemble listing.

The **cross\_reference** and **nocross\_reference** options are valid only if an assemble listing is being output.

## Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result (When an Assemble Listing Is Output)
cross_reference	(regardless of any specification)	A cross-reference listing is output.
nocross_reference	(regardless of any specification)	A cross-reference listing is not output.
(no specification)	.PRINT CREF	A cross-reference listing is output.
	.PRINT NOCREF	A cross-reference listing is not output.
	(no specification)	A cross-reference listing is output.

## S~~E~~ction, NO~~S~~Ection

Assembly <List > [Section :]

- Command Line Format

S~~E~~ction

NO~~S~~Ection

- Description

The **section** option specifies output of a section information listing to the assemble listing.

The **nosection** option specifies no output of a section information listing to the assemble listing.

The **section** and **nosection** options are valid only if an assemble listing is being output.

### Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result (When an Assemble Listing Is Output)
section	(regardless of any specification)	A section information listing is output.
nosection	(regardless of any specification)	A section information listing is not output.
(no specification)	.PRINT SCT	A section information listing is output.
	.PRINT NOSCT	A section information listing is not output.
	(no specification)	A section information listing is output.

### 3.2.4 Tuning Options

Table 3.4 Tuning Options

Item	Command Line Format	Dialog Menu	Specification
Specification of symbols for 8- or 16-bit absolute address format	ABS8 ABS16	Assembly <Tuning> [Option to set :]	Specifies whether symbols to be accessed as 8- or 16-bit absolute addresses.

#### ABS8, ABS16

Assembly <Tuning> [Option to set :]

- Command Line Format  
ABS8 [ = <symbol> [,...] ]  
ABS16 [ = <symbol> [,...] ]

- Description

The **abs8** option specifies a symbol to be accessed in 8-bit absolute address format.

The **abs16** option specifies a symbol to be accessed in 16-bit absolute address format.

When a symbol is omitted, all externally referenced symbols and externally defined symbols are specified.

When the **abs8** and **abs16** options are both specified for the same symbol, the option on the right hand side has the priority:

— When **–abs8 –abs16** is specified:

All external symbols are accessed in 16-bit absolute address format.

— When **–abs8=<symbol> –abs16=<symbol>** is specified:

<symbol> is accessed in 16-bit absolute address format, and all others are determined by the CPU. However, when a symbol is specified in one option and when symbols are omitted in another option, both options are exclusively valid.

— When **–abs8=<symbol> –abs16** is specified:

<symbol> is accessed in 8-bit absolute address format, and the others are accessed in 16-bit absolute address format.

Priority of Access Size Settings



Priority	Access Size Format
High	1 Size specified for the absolute address format
	2 Access size set by .IMPORT, .EXPORT, or .GLOBAL directives .ABS8 and .NOABS8 directives
Low	3 abs8 or abs16 settings

Example: `asm38 aaa.mar -abs8=sym1 -abs16`

When an external symbol is specified in the absolute address format, sym1 is addressed in 8-bit absolute address format, and other external symbols are addressed in 16-bit absolute address format.

`asm38 aaa.mar-abs8=sym1 -abs16=sym2,sym3,sym4`

Contents of `aaa.mar`

```
.CPU      2600A
.IMPORT   sym1,sym2,sym3,sym5
.IMPORT   sym4:8
MOV.B    @sym1    ,R1H      ;8 bits  (-abs8 option specified)
MOV.B    @sym2    ,R1H      ;16 bits (-abs16 option specified)
MOV.B    @sym3:8  ,R1H      ;8 bits  (size explicitly specified)
MOV.B    @sym4    ,R1H      ;8 bits  (address size specified by .IMPORT)
MOV.B    @sym5    ,R1H      ;32 bits (no specification)
MOV.B    @(sym1+sym2),R1H    ;8 bits  (the smaller of -abs8 and -abs16)
```

Note: When more than one external symbols is specified for the absolute address format, the minimum address size is used.

### 3.2.5 Other Options

Table 3.5 Other Options

Item	Command Line Format	Dialog Menu	Specification
Unreferenced import symbol output control	Exclude  <u>NOExclude</u>	Assembly <Other> [Miscellaneous options :] [Remove unreferenced external symbols]	Not output the symbol information on import symbols that have not been referred to. Outputs the symbol information on import symbols that have not been referred to.

#### Exclude, NOExclude

Assembly <Other> [Miscellaneous options :] [Remove unreferenced external symbols]

- Command Line Format

Exclude

NOExclude

- Description

The **exclude** option prevents the output of symbol information on import symbols that have not been referred to.

The **noexclude** option specifies the output of the symbol information on import symbols that have not been referred to.

Suppressing the output of this information makes the object modules smaller.

Example: `asm38 aaa.mar -exclude`

The information on import symbols that have not been referred to is not output.

`asm38 aaa.mar -noexclude`

The information on import symbols that have not been referred to is output.

### 3.2.6 CPU Options

**Table 3.6 CPU Options**

Item	Command Line Format	Dialog Menu	Specification
CPU type specification	CPU = { AE5 H8SXN[:{M D MD}] H8SXM[:<bit width>] [:{M D MD}] H8SXA[:<bit width>] [:{M D MD}] H8SXX[:<bit width>] [:{M D MD}] 2600N 2600A [:<bit width>] 2000N 2000A [:<bit width>] 300HN 300HA [:<bit width>] 300   300L	CPU [CPU :]	Specifies the CPU type.
Origin specification in the 8-bit short absolute area	SBR	CPU [Specify SBR address :]	Specifies the origin of the 8-bit short absolute area.

## CPu

CPU [CPU :]

- Command Line Format

```
CPu = {AE5 |  
      H8SXN [ :{M|D|MD}] |  
      H8SXM [ :<bit width of the address space>] [ :{M|D|MD}] |  
      H8SXA [ :<bit width of the address space>] [ :{M|D|MD}] |  
      H8SXX [ :<bit width of the address space>] [ :{M|D|MD}] |  
      2600N |  
      2600A [ :<bit width of the address space>] |  
      2000N |  
      2000A [ :<bit width of the address space>] |  
      300HN |  
      300HA [ :<bit width of the address space>] |  
      300 | 300L } }
```

- Description

Specifies the CPU type and the operating mode for the object program to be generated, the bit width of the address space, and whether or not a multiplier and/or a divider exist.

Table 3.7 lists the suboptions.

**Table 3.7 Suboptions for cpu Option**

<b>Suboption</b>	<b>Description</b>
AE5	Creates an object for the AE5. Refer to section 17, Supporting AE5 Features.
H8SXN[:{M D MD}]	Creates an object for the H8SX normal mode. A multiplier and/or a divider can be specified.
H8SXM[:<bit width of the address space>][:{M D MD}]	Creates an object for the H8SX middle mode. <bit width of the address space> is 20 or 24, which is 1 Mbyte or 16 Mbytes, respectively. <bit width of the address space> is 24 by default. A multiplier and/or a divider can be specified.
H8SXA[:<bit width of the address space>][:{M D MD}]	Creates an object for the H8SX advanced mode. <bit width of the address space> is 20, 24, 28, or 32, which is 1 Mbyte, 16 Mbytes, 256 Mbytes, or 4 Gbytes, respectively. <bit width of the address space> is 24 by default. A multiplier and/or a divider can be specified.
H8SXX[:<bit width of the address space>][:{M D MD}]	Creates an object for the H8SX maximum mode. <bit width of the address space> is 28 or 32, which is 256 Mbytes or 4 Gbytes, respectively. <bit width of the address space> is 32 by default. A multiplier and/or a divider can be specified.
2600N	Creates an object for the H8S/2600 normal mode.
2600A[:<bit width of the address space>]	Creates an object for the H8S/2600 advanced mode. The value of <bit width of the address space> is 20, 24, 28, or 32, to indicate 1 Mbyte, 16 Mbytes, 256 Mbytes, or 4 Gbytes, respectively. <bit width of the address space> is 24 by default.
2000N	Creates an object for the H8S/2000 normal mode.
2000A[:<bit width of the address space>]	Creates an object for the H8S/2000 advanced mode. The value of <bit width of the address space> is 20, 24, 28, or 32, to indicate 1 Mbyte, 16 Mbytes, 256 Mbytes, or 4 Gbytes, respectively. <bit width of the address space> is 24 by default.
300HN	Creates an object for the H8/300H normal mode.
300HA[:<bit width of the address space>]	Generates the object for the H8/300H advanced mode. The value of <bit width of the address space> is 20 or 24, to indicate 1 Mbyte or 16 Mbytes, respectively. <bit width of the address space> is 24 by default.
300	Creates an object for the H8/300.
300L	Creates an object for the H8/300L.

Specify whether or not a multiplier and a divider exist as follows:

Multiplier/Divider	Specification Method
Without multiplier and without divider	No specification
With multiplier and without divider	M
Without multiplier and with divider	D
With multiplier and with divider	MD

Use MAC, LDMAC, STMAC, CLRMAC, MULU/U, or MULS/U as an additional instruction with a multiplier.

There are no additional instructions with a divider.

- Remarks

When the **cpu** option is omitted, the contents of the H38CPU environmental variable are referred to. Priority is given to the **cpu** option when both a **cpu** option and H38CPU environmental variable are specified. When neither a **cpu** option nor a H38CPU environmental variable is set, the error message 933 is output.

Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Environmental Variable	Result (When an Assemble Listing Is Output)
cpu=cpu type	(regardless of any specification)	(regardless of any specification)	Cpu type as specified by cpu.
(no specification)	.CPU	cpu type	Cpu type as specified by the .CPU.
	(no specification)	h38cpu=cpu type	Cpu type set by the environmental variable.
		(no specification)	Output of error message 933

## SBR

CPU [Specify SBR address :]

- Command Line Format  
SBR = {<constant> | USER}
- Description

When SBR=<constant> is specified, the 256-byte area whose origin is <constant> as the access area of the 8-bit absolute addressing format. As for the constant, radix H' should be specified and the lower 8 bits should be fixed to 0. When SBR = USER is specified, the origin of the 8-bit short absolute address is as shown below depending on the bit width of the address space.

CPU/Operating Mode	Origin of the 8-Bit Short Absolute Address	
H8SX maximum mode	H8SXX[:32]	H'FFFFFF00
	H8SXX:28	H'0FFFFFF00
H8SX advanced mode	H8SXA:32	H'FFFFFF00
	H8SXA:28	H'0FFFFFF00
	H8SXA[:24]	H'00FFFF00
	H8SXA:20	H'000FFF00
H8SX middle mode	H8SXM[:24]	H'00FFFF00
	H8SXM:20	H'000FFF00
H8SX normal mode	H8SXN	H'0000FF00

Only when the CPU is H8SXN, H8SXM, H8SXA, or H8SXX, the SBR option can be specified.

## Relationship with Assembler Directives

Option	Assembler Directive	Origin of the Access Area of the 8-Bit Absolute Address
sbr=<constant>	.SBR <constant>	Constant specified with the SBR directive
	.SBR	Constant specified with the sbr option
	(no specification)	Constant specified with the sbr option
sbr=USER	.SBR <constant>	Constant specified with the SBR directive
	.SBR	Value determined by the bit width of the address space
	(no specification)	Value determined by the bit width of the address space
(no specification)	.SBR <constant>	Constant specified with the SBR directive
	.SBR	Value determined by the bit width of the address space
	(no specification)	Value determined by the bit width of the address space

Example: `asm38 aaa.mar -sbr=H'ff0000`

The 8-bit short absolute address area is in the range from H'00ff0000 to H'00ff00ff.

Contents of aaa.mar

`.CPU H8SXX:32`

`MOV.L #H'00ff0000,ER1`

`LDC.L ER1,SBR`

`MOV.B @sym1 ,R1H` ;8 bits (within the 8-bit short absolute  
; address area specified with -sbr)

`MOV.B @sym2 ,R1H` ;16 bits (without the 8-bit short absolute  
; address area specified with -sbr)

`sym1: .equ H'00ff0040`

`sym2: .equ H'ffffff40`

- Remarks

If the host computer OS is UNIX, specify a backslash (\) before the apostrophe (') of the radix indicator "H".



### 3.2.7 Options Other Than Above

**Table 3.8 Options Other Than Above**

Item	Command Line Format	Dialog Menu	Specification
Change of error level at which the assembler is abnormally terminated	ABort = {Warning   <u>Error</u> }	Assembly <Other> [User defined options :]	Changes the error level at which the assembler is abnormally terminated.
ISO-Latin1 Code	LATIN1	Assembly <Other> [User defined options :]	Enables the use of Latin1 code characters in source file.
Shift JIS code	SJIS	Assembly <Other> [User defined options :]	Interprets Japanese character in source file as shift JIS code.
EUC code	EUC	Assembly <Other> [User defined options :]	Interprets Japanese character in source file as EUC code.
Specification of Japanese character	OUTcode = {SJIS   EUC}	Assembly <Other> [User defined options :]	Specifies the Japanese character for output to object code.
Setting of the number of lines in the assemble listing	LINEs = <number of lines>	Assembly <Other> [User defined options :]	Specifies the number of lines in assemble listing.
Setting of the number of digits in the assemble listing	COLUMNs = <number of digits>	Assembly <Other> [User defined options :]	Specifies the number of digits in assemble listing.
Copyright	<u>LOGO</u> NOLOGO	- (nologo is always valid)	Outputs logo Not output logo
Specification of subcommand	SUBcommand = <file name>	-	Inputs command line from a file.

## ABort

Assembly <Other> [User defined options :]

- Command Line Format  
ABort = { Warning | Error }

- Description

The **abort** option specifies the error level.

When the return value to the OS becomes 1 or larger, the object module is not output.

The **abort** option is valid only if the object module is output.

The return value to the OS is as follows:

Number of Cases			Return Value to OS when Option Specified			
			abort=warning		abort=error	
Warning	Error	Fatal Error	PC	UNIX	PC	UNIX
0	0	0	0	0	0	0
1 or more	0	0	2	1	0	0
—	1 or more	0	2	1	2	1
—	—	1 or more	4	1	4	1

## LATIN1

Assembly <Other> [User defined options :]

- Command Line Format  
LATIN1

- Description

The **latin1** option enables the use of ISO-Latin1 code characters in strings literal and in comments.

Do not specify this option together with the **sjis**, **euc**, or **outcode** option.

## SJIS

Assembly <Other> [User defined options :]

- Command Line Format

SJIS

- Description

When the **sjis** option is specified, Japanese characters in strings literal and comments are interpreted as shift **JIS** code.

When the **sjis** option is omitted, Japanese characters in strings literal and comments are interpreted as Japanese characters depending on the host computer.

Do not specify this option together with the **latin1** or **euc** option.

## EUC

Assembly <Other> [User defined options :]

- Command Line Format

EUC

- Description

When the **euc** option is specified, Japanese characters in strings literal and comments are interpreted as **EUC** code.

When the **euc** option is omitted, Japanese characters in strings literal and comments are interpreted as Japanese characters depending on the host computer.

Do not specify this option together with the **latin1** or **sjis** option.

## OUtcode

Assembly <Other> [User defined options :]

- Command Line Format  
OUtcode = {SJIS | EUC}

- Description

The **outcode** option converts Japanese characters in the source file to the specified Japanese character for output to the object file.

The Japanese character output to the object file depends on the **outcode** specification and the Japanese character (**sjis** or **euc**) in the source file as follows:

outcode Specification	Japanese Character in Source File		
	sjis	euc	No Specification
sjis	Shift JIS code	Shift JIS code	Shift JIS code
euc	EUC code	EUC code	EUC code
No specification	Shift JIS code	EUC code	Default code

Default code is as follows.

Host Computer	Default Code
PC	Shift JIS code
SPARC station	EUC code
HP9000/700 series	Shift JIS code

## LINes

Assembly <Other> [User defined options :]

- Command Line Format  
LINes = <Number of lines>

- Description

The **lines** option sets the number of lines on a single page of the assemble listing. The range of valid values for the line count is from 20 to 255.

The **lines** option is valid only if an assemble listing is being output.

### Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result
lines=<number of lines>	(regardless of any specification)	The number of lines on a page is given by lines
(no specification)	.FORM LIN=<number of lines>	The number of lines on a page is given by .FORM.
	(no specification)	The number of lines on a page is 60 lines.

## Columns

Assembly <Other> [User defined options :]

- Command Line Format  
Columns = <Number of digits>

- Description

The **columns** option sets the number of digits in a single line of the assemble listing. The range of valid values for the column count is from 79 to 255.

The **columns** option is valid only if an assemble listing is being output.

## Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result
columns= <number of digits>	(regardless of any specification)	The number of digits in a line is given by columns.
(no specification)	.FORM COL=<number of digits>	The number of digits in a line is given by .FORM.
	(no specification)	The number of digits in a line is 132.

## LOGO, NOLOGO

None (nologo is always available)

- Command Line Format

LOGO

NOLOGO

- Description

Disables the copyright output.

When the **logo** is specified, copyright display is output.

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

When this option is omitted, **logo** is assumed.

## SUBcommand

None

- Command Line Format

SUBcommand = <file name>

- Description

The **subcommand** option inputs command line specifications from a file.

Specify input file names and options in the subcommand file in the same order as for normal command line specifications.

Only one input file name or option can be specified in one line in the subcommand file.

This option must not be specified in a subcommand file.

Example:     asm38 aaa.src -subcommand=aaa.sub

The subcommand file contents are expanded to a command line and assembled.

Contents of aaa.sub

```
bbb.src
```

```
-list
```

```
-noobj
```

The above command line and file aaa.sub are expanded as follows:

```
asm38 aaa.src,bbb.src -list -noobj
```

- Remarks

One subcommand file can include a maximum of 65,535 bytes.





# Section 4 Optimizing Linkage Editor Options

## 4.1 Option Specifications

### 4.1.1 Command Line Format

The format of the command line is as follows:

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

### 4.1.2 Subcommand File Format

The format of the subcommand file is as follows:

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

For details, refer to section 4.2.8, Subcommand File Option.

## 4.2 List of Options

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

The format of the dialog menus that correspond to the HEW is as follows:

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

For details on dialog menus, refer to the HEW.

The order of option description corresponds to that of the tabs and the categories in the HEW.

## 4.2.1 Input Options

**Table 4.1 Input Category Options**

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

### Input

### Input File

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

- Command Line Format  
Input = <suboption>[{, | Δ}...]  
<suboption>: <file name>[ (<module name>[,...]) ]
- Description  
Specifies an input file. Two or more files can be specified by separating them with a comma (,) or space.

Wildcards (\* or ?) can also be used for the specification. String literals specified with wildcards are expanded in alphabetical order. Expansion of numerical values precedes that of alphabetical letters. Uppercase letters are expanded before lowercase letters.

Specifiable files are object files output from the compiler or the assembler, and relocatable or absolute files output from the optimizing linkage editor. A module in a library can be specified as an input file using the format of **<library name>(<module name>)**. The module name is specified without an extension.

If an extension is omitted from the input file specification, **obj** is assumed when a module name is not specified and **lib** is assumed when a module name is specified.

- Example

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

- Remarks

When **form=object** or **extract** is specified, this option is unavailable.

When an input file is specified on the command line, **input** should be omitted.

## LIBrary

## Library File

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

- Command Line Format

LIBrary = <file name>[,...]

- Description

Specifies an input library file. Two or more files can be specified by separating them with a comma (,).

Wildcards (\* or ?) can also be used for the specification. String literals specified with wildcards are expanded in the alphabetical order. Expansion of numerical values precedes that of alphabetical letters. Uppercase letters are expanded before lowercase letters.

If **form=library** or **extract** is specified, the library file is input as the target library to be edited.

Otherwise, after the linkage processing between files specified for the input files are executed, undefined symbols are searched in the library file.

The symbol search in the library file is executed in the following order: user library files with the library option specification (in the specified order), the system library files with the library option specification (in the specified order), and then the default library (environment variable **HLNK\_LIBRARY1,2,3**).

- Example

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

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

- Command Line Format

Binary = <suboption>[,...]

<suboption>: <file name>(<section name>[:<boundary alignment>][,<symbol name>])

<boundary alignment>: 1 | 2 | 4 | 8 | 16 | 32 (default: 1)

- Description

Specifies an input binary file. Two or more files can be specified by separating them with a comma (.).

If an extension is omitted for the file name specification, **bin** is assumed.

Input binary data is allocated as the specified section data. The section address is specified with the **start** option. That section cannot be omitted.

When a symbol is specified, the file can be linked as a defined symbol. For a variable name referenced by a C/C++ program, add an underscore (\_) at the head of the reference name in the program.

A boundary alignment value can be specified for the section specified by this option. A power of 2 can be specified for the boundary alignment; no other values should be specified.

When the boundary alignment specification is omitted, 1 is used as the default.

- Example

```
input=a.obj
```

```
start=P,D*/200
```

```
binary=b.bin(D1bin),c.bin(D2bin:4,_datab)
```

Allocates **b.bin** from **0x200** as the **D1bin** section.

Allocates **c.bin** after **D1bin** as the **D2bin** section (with boundary alignment = 4).

Links **c.bin** data as the defined symbol **\_datab**.

- Remarks

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

If no input object file is specified, this option cannot be specified.

## DEFine

## Symbol Definition

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

- Command Line Format

DEFine = <suboption>[,...]

<suboption>: <symbol name>={<symbol name> | <numerical value>}

- Description

Defines an undefined symbol forcedly as an externally defined symbol or a numerical value.

The numerical value is specified in the hexadecimal notation. If the specified value starts with a letter from A to F, symbols are searched first, and if no corresponding symbol is found, the value is interpreted as a numerical value. Values starting with 0 are always interpreted as numerical values.

If the specified symbol name is a **C/C++** variable name, add an underscore (**\_**) at the head of the definition name in the program. If the symbol name is a **C++** function name (except for the main function), enclose the definition name with the double-quotation marks including parameter strings. If the parameter is void, specify as "<function name>()".

- Example

```
define=_sym1=data      ;Defines _sym1 as the same value as the externally defined
                        symbol data.
```

```
define=_sym2=4000      ;Defines _sym2 as 0x4000.
```

- Remarks

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

## ENTry

## Execution Start Address

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

- Command Line Format

```
ENTry = {<symbol name> | <address>}
```

- Description

Specifies the execution start address with an externally defined symbol or address.

The address is specified in hexadecimal notation. If the specified value starts with a letter from A to F, symbols are searched first, and if no corresponding symbol is found, the value is interpreted as an address. Values starting with 0 are always interpreted as addresses.

For a **C** function name, add an underscore (**\_**) at the head of the definition name in the program. For a **C++** function name (except for the **main** function), enclose the definition name with double quotation marks in the program including parameter strings. If the parameter is void, specify as "<function name>()".

If the **entry** symbol is specified at compilation or assembly, this option precedes the **entry** symbol.

- Example

```
entry=_main           ; Specifies main function in C/C++ as the execution start address.
```

```
entry="init()"        ; Specifies init function in C++ as the execution start address.
```

```
entry=100             ; Specifies 0x100 as the execution start address.
```

- Remarks

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

When optimization with undefined symbol deletion (**optimize=symbol\_delete**) is specified, the execution start address should be specified. If it is not specified, the specification of the optimization with undefined symbol deletion is unavailable.

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

- Command Line Format

NOPRElink

- Description

Disables the prelinker initiation.

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

- Remarks

When **extract** or **strip** is specified, this option is unavailable.

## 4.2.2 Output Options

**Table 4.2 Output Category Options**

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

**Table 4.2 Output Category Options (cont)**

Item	Command Line Format	Dialog Menu	Specification
Information message	Message <u>NOMessage</u> [= <sub>[,...]] <sub>:<error code> [-<error code>]	Link/Library <Output> [Show entries for :] [Output file path/ Messages] [Repressed information level messages:]	Output No output (error number specification and range specification are enabled)
Notification of unreferenced defined symbol	MSg_unused	Link/Library <Output> [Show entries for :] [Notify unused symbol:]	Notifies the user of the defined symbol which is never referenced
Reduce empty areas of boundary alignment	DAta_stuff	Link/Library <Output> [Show entries for :] [Reduce empty areas of boundary alignment:]	Reduces empty areas generated as the boundary alignment of sections after compilation

**Form****Output Format**

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

- Command Line Format

Form = { Absolute | Relocate | Object | Library[={S | U}] }  
          | Hexadecimal | Stype | Binary }

- Description

Specifies the output format.

When this option is omitted, the default is **form=absolute**. Table 4.3 lists the suboptions.



**Table 4.3 Suboptions of Form Option**

Suboption	Description
absolute	Outputs an absolute file
relocate	Outputs a relocatable file
object	Outputs an object file. This is specified when a module is extracted as an object file from a library with the <b>extract</b> option.
library	Outputs a library file. When <b>library=s</b> is specified, a system library is output. When <b>library=u</b> is specified, a user library is output. Default is <b>library=u</b> .
hexadecimal	Outputs a HEX file. For details of the HEX format, refer to appendix 19.1.2, HEX File Format.
stype	Outputs an S-type file. For details of the S-type format, refer to appendix 19.1.1, S-Type File Format.
binary	Outputs a binary file.

- Remarks

Table 4.4 shows relations between output formats and input files or other options.

**Table 4.4 Relations Between Output Format And Input File Or Other Options**

<b>Output Format</b>	<b>Specified Option</b>	<b>Enabled File Format</b>	<b>Specifiable Option*<sup>1</sup></b>
Absolute	strip specified	Absolute file	input, output, hide, show=symbol, reference
	other than above	Object file Relocatable file Binary file Library file	input, library, binary, debug/nodebug, sdebug, cpu, start, rom, entry, output, map, hide, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, absolute_forbid, profile, cachesize, sbr, compress, rename, delete, define, fsymbol, stack, noprelink, memory, msg_unused, data_stuff, show=symbol, reference, xreference,
Relocate	extract specified	Library file	library, output, show=symbol, reference
	other than above	Object file Relocatable file Binary file Library file	input, library, debug/nodebug, output, hide, rename, delete, noprelink, msg_unused, data_stuff, show=symbol, reference, xreference.
Object	extract specified	Library file	Library, output, show=symbol
Relocate Stype Binary		Object file Relocatable file Binary file Library file	Input, library, binary, cpu, start, rom, entry, output, map, space, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, absolute_forbid, profile, cachesize, sbr, rename, delete, define, fsymbol, stack, noprelink, record, s9* <sup>2</sup> , memory, msg_unused, data_stuff, show=symbol, reference, xreference
		Absolute file	input, output, record, s9* <sup>2</sup> , show=symbol, reference, xreference
Library	strip specified	Library file	library, output, hide, show=symbol, section
	extract specified	Library file	library, output, show=symbol, section
	other than above	Object file Relocatable file	input, library, output, hide, rename, delete, replace, noprelink, show=symbol, section

- Notes: 1. **message/nomessage, change\_message, logo/nologo, form, list,** and **subcommand** can always be specified.  
2. s9 can be used only when **form=stype** is specified for the output format.

Link/Library <Output>[Debug information :]

- Command Line Format

DEBug

SDeBug

NODeBug

- Description

Specifies whether debug information is output.

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

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

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

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

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

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

- Remarks

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

## REcord

## Record Size Unification

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

- Command Line Format

Record = { H16 | H20 | H32 | S1 | S2 | S3 }

- Description

Outputs data with the specified data record regardless of the address range.

If there is an address that is larger than the specified data record, the appropriate data record is selected for the address.

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

- Remarks

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

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

- Command Line Format  
ROm = <suboption>[,...]  
<suboption>: <ROM section name>=<RAM section name>
- Description  
Reserves ROM and RAM areas in the initialized data area and relocates a defined symbol in the ROM section with the specified address in the RAM section.  
Specifies a relocatable section including the initial value for the ROM section.  
Specifies a nonexistent section or relocatable section whose size is 0 for the RAM section.
- Example  
rom=D=R  
start=D/100,R/8000  
Reserves **R** section with the same size as **D** section and relocates defined symbols in **D** section with the **R** section addresses.
- Remarks  
When **form**=**{ object | relocate | library }** or **strip** is specified, this option is unavailable.

## OUtput

## Output File

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

- Command Line Format  
OUtput = <suboption>[,...]  
<suboption>: <file name>[=<output range>]  
<output range>: { <start address>-<end address> | <section name>[:...]}
- Description  
Specifies an output file name. When **form**=**absolute**, **hexadecimal**, **stype** or **binary** is specified, two or more files can be specified. An address is specified in the hexadecimal notation. If the specified data starts with a letter from A to F, sections are searched first, and if no corresponding section is found, the data is interpreted as an address. Data starting with 0 are always interpreted as addresses.  
When this option is omitted, the default is <first input file name>.<default extension>.  
The default extensions are as follows:  

form=absolute: abs	form=relocate: rel	form=object: obj
form=library: lib	form=hexadecimal: hex	form=stype: mot
form=binary: bin		

- Example

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

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

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

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

## MAp

## Output of External Symbol Allocation Information File

Link/Library <Output>[Generate map file]

- Command Line Format

MAp [= <file name>]

- Description

Outputs the external-symbol-allocation information file that is used by the compiler in optimizing access to external variables.

When <file name> is not specified, the file has the name specified by the **output** option or the name of the first input file, and the extension **bls**.

If the order of the declaration of variables in the external-symbol-allocation information file is not the same as the order of the declaration of variables found when the object was read after compilations, an error will be output.

- Remarks

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

## SPace

## Output to Unused Areas

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

- Command Line Format

SPace [= <numerical value> ]

- Description

Specifies a hexadecimal value to fill the unused areas in the output range.

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

When section names are specified for the output range:

The specified value is output to unused areas between the specified sections.

When an address range is specified for the output range:

The specified value is output to unused areas within the specified address range.

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

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

- Remarks

When no numerical value is specified by this option, unused areas are not filled with values.

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

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

## Message, NOMessage

## Information Message

Link/Library <Output>[Show entries for :] [Output file path/ Messages]

[Repressed information level messages :]

- Command Line Format

Message

NOMessage [=<suboption>[,...]]

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

- Description

Specifies whether information level messages are output.

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

When **nomessage** is specified, the output of information level messages are disabled. If an error number is specified, the output of the error message with the specified error number is disabled. A range of error message numbers to be disabled can be specified using a hyphen (-). If a warning or error level message number is specified, the message output is disabled assuming that **change\_message** has changed the specified message to the information level.

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

- Example

`nomessage=4,200-203,1300`

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

## **MSg\_unused**

## **Notification of Unreferenced Symbol**

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

- Command Line Format

`MSg_unused`

- Description

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

- Example

`optlnk -msg_unused a.obj`

- Remarks

When an absolute file is input, this option is invalid.

To output a notification message, the **message** option must also be specified.

In any of the following cases, references are not correctly analyzed so that information shown by output messages will be incorrect.

— **-goptimize** is not specified at assembly and there are branches to the same section within the same file (only when an H8-series CPU is specified).

— There are references to constant symbols within the same file.

— There are branches to immediate subordinate functions when optimization is specified at compilation.

— The **map** optimization is valid at compilation (only when an SH-series CPU is specified).

— An offset value is directly specified in a **#pragma tbr** in the C source program (only when **sh2a** or **sh2afpu** is specified as the CPU).

— Optimization is specified at linkage and constants or literals are unified.

## **DAta\_stuff**

## **Reduce empty areas of boundary alignment**

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

- Command Line Format

`DAta_stuff`

- Description

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

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

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

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

- Example

<code>&lt;tp1.c&gt;</code>	<code>&lt;tp2.c&gt;</code>
<code>long a;</code>	<code>char d;</code>
<code>char b,c;</code>	<code>long e;</code>
	<code>char f;</code>

Sizes of data sections after compilation (taking the output of the SH compiler as an example):

tp1.obj : 4 + 1 + 1 = 6 bytes

tp2.obj : 1 + 3 [\*] + 4 + 1 = 9 bytes

Sizes of data sections for tp1.obj and tp2.obj after linkage:

- When **data\_stuff** is not specified

Object files are linked based on the boundary alignment of the sections (conventional process).

6 bytes [tp1] + 2 bytes [\*] + 9 bytes [tp2] = 17 bytes

- When **data\_stuff** is specified

Linkage is performed with filling of the unnecessary empty spaces generated between sections by boundary alignment.

(4 + 1 + 1) bytes + 1 byte + 1 byte [\*] + 4 bytes + 1 byte = 13 bytes

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

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

- Remarks

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

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

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

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



## 4.2.3 List Options

**Table 4.5 List Category Options**

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

### LISt

### List File

Link/Library <List>[Generate list file]

- Command Line Format

LISt [=<file name>]

- Description

Specifies list file output and a list file name.

If no list file name is specified, a list file with the same name as the output file (or first output file) is created, with the extension **lb**p when **form=library** or **extract** is specified, or **map** in other cases.

### SHow

### List Contents

Link/Library <List>[Contents :]

- Command Line Format

SHow [=<suboption>[,...]]

<suboption>: { SYmbol | Reference | SEction | Xreference }

- Description

Specifies output contents of a list.

Table 4.6 lists the suboptions.

For details of list examples, refer to section 8.4, Linkage Listings, and section 8.5, Library Listings.

**Table 4.6 Suboptions of show Option**

Output Format	Suboption Name	Description
form=library or extract is specified.	symbol	Outputs a symbol name list in a module
	reference	Cannot be specified
	section	Outputs a section list in a module
	xreference	Cannot be specified
Other than form=library and extract is not specified.	symbol	Outputs symbol address, size, type, and optimization contents.
	reference	Outputs the number of symbol references
	section	Cannot be specified
	xreference	Outputs the cross-reference information

- Remarks

When **form={ object | relocate }** is specified, the **show=reference** option is invalid.

When **form=library** is specified, the **show=xreference** option is invalid.

When outputting the cross-reference information, note the following limitations.

- When an absolute-format file is input, the referrer address information is not output.
- When **-goptimize** is not specified at assembly, information about branches to the same section within the same file is not output (only when an H8 CPU is specified).
- Information about references to constant symbols within the same file is not output.
- When optimization is specified at compilation, information about branches to immediate subordinate functions is not output.
- When the **map** optimization is specified, information about references to variables other than base symbols is not output (only when an SH-series CPU is specified).
- When an offset value is directly specified in a **#pragma thr** in the C source program, information about that function is not output (only when **sh2a** or **SH2AFPU** is specified as the CPU).
- When optimization is specified at linkage and constants or literals are unified, information about references to these constants or literals is not output.

## 4.2.4 Optimize Options

**Table 4.7 Optimize Category Options**

Item	Command Line Format	Dialog Menu	Specification
Optimization	<b>Optimize</b> = <sub>[...] <sub>: {SString_unify}   SYmbol_delete   Variable_access   Register   SAME_code   SHort_format   Function_call   Branch   SPeed   SAFe <b>NOOptimize</b>	Link/Library <Optimize> [Show entries for :] [Optimize items] [Optimize :]	Executes optimization. Unifies constants/string literals. Deletes unreferenced symbols. Uses short absolute addressing mode. Provides optimization with register save/restore. Unifies same codes. Shortens the addressing mode. Uses indirect addressing mode. Provides optimization for branches. Provides optimization for speed. Provides safe optimization. No optimization.
Same code size	<b>SAMESize</b> = <size> (default: <u>sames=1e</u> )	Link/Library <Optimize> [Eliminated size :]	Specifies the minimum size to unify same codes.
Profile information	<b>PROfile</b> = <file name>	Link/Library <Optimize> [Include profile :]	Specifies a profile information file. (Dynamic optimization is provided.)
Cache size	<b>CAchesize</b> =<sub> <sub>: Size=<size>   Align=<line size> (default: <u>ca=s=8,a=20</u> )	Link/Library <Optimize> [Cache size :]	Specifies a cache size. Specifies a cache line size.
Optimization partially disabled	<b>SYmbol_forbid</b> = <symbol name>[,...] <b>SAMECode_forbid</b> = <function name>[,...] <b>Variable_forbid</b> = <symbol name>[,...] <b>FUnction_forbid</b> = <function name>[,...] <b>Absolute_forbid</b> = <address>[+<size>][,...]	Link/Library <Optimize> [Show entries for :] [Forbid item]	Specifies a symbol where unreferenced symbol deletion is disabled. Specifies a symbol where same code unification is disabled. Specifies a symbol where short absolute addressing mode is disabled. Specifies a symbol where indirect addressing mode is disabled. Specifies an address range where optimization is disabled.

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

- Command Line Format

OPTimize [= <suboption>[,...]]

NOOPTimize

<suboption>: { SString\_unify | SYmbol\_delete | Variable\_access | Register | SAME\_code |  
SHort\_format | Function\_call | Branch | SPeed | SArithmetic }

- Description

Specifies whether the inter-module optimization is executed.

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

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

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

Table 4.8 shows the suboptions

**Table 4.8 Suboptions of Optimize Option**

Suboption	Description	Program to be Optimized <sup>1</sup>			
		SHC	SHA	H8C	H8A
No parameter	Provides all optimizations	O	X	O	O
string_unify	Unifies same-value constants having the const attribute. Constants having the const attribute are: <ul style="list-style-type: none"> <li>• Variables defined as const in C/C++ program</li> <li>• Initial value of character string data</li> <li>• Literal constant</li> </ul>	O	X	O	X
symbol_delete	Deletes variables/functions that are not referenced. The <b>entry</b> option should be specified.	O	X	O	X
variable_access	Allocates frequently accessed variables to the area accessible in the 8/16 bit absolute addressing mode. The <b>cpu</b> option should be specified.	X	X	O	O
register	Investigates function calls, relocates registers and deletes redundant register save or restore codes. The <b>entry</b> option should be specified.	O	X	O	X
same_code	Creates a subroutine for the same instruction sequence.	O	X	O	X
short_format	Replaces an instruction having a displacement or an immediate value with a smaller-size instruction when the code size of the displacement or immediate value can be reduced.	X	X	O	O

**Table 4.8 Suboptions of Optimize Option (cont)**

Suboption	Description	Program to be Optimized <sup>1</sup>			
		SHC	SHA	H8C	H8A
function_call	Allocates addresses of frequently accessed functions to the range 0 to 0xFF if there is a space. When the CPU is H8SX, the following ranges are also used: H8SXN: 0x100 to 0x1FF H8SXM,H8SXA,H8SXX: 0x200 to 0x3FF The <b>cpu</b> option should be specified.	X	X	O	O
branch	Optimizes branch instruction size according to program allocation information. Even if this option is not specified, it is performed when any other optimization is executed.	O	X	O	O
speed	Executes optimizations other than those reducing object speed. This suboption is the same as the following specifications: optimize=string_unify, symbol_delete, variable_access, register, short_format, or branch	O	X	O	O
safe	Executes optimizations other than those limited by variable or function attributes. This suboption is the same as the following specifications: optimize=string_unify, register, short_format, or branch	O	X	O	O

Note: SHC: C/C++ program for SH  
SHA: Assembly program for SH  
H8C: C/C++ program for H8  
H8A: Assembly program for H8

- Remarks

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

When map optimization is specified at compilation, unifies constants/string literal optimization (**optimize=string\_unify**) is invalid.

**optimize=short\_format** is available only when the CPU is H8SX.

## SAMESize

## Common Code Size

Link/Library <Optimize>[Eliminated size :]

- Command Line Format

SAMESize = <size>

- **Description**  
Specifies the minimum code size for the optimization with the same-code unification (**optimize=same\_code**). Specify a hexadecimal value from 8 to 7FFF.  
When this option is omitted, the default is **samesize=1E**.
- **Remarks**  
When **optimize=same\_code** is not specified, this option is unavailable.

## PROfile

## Profile Information

Link/Library <Optimize>[Include profile :]

- **Command Line Format**  
PROfile = <file name>
- **Description**  
Specifies a profile information file.  
Specifiable profile information files are those output from the Hitachi Debugging Interface Ver. 5.0 or later or from the HEW Ver. 2.0 or later.  
When a profile information file is specified, inter-module optimization according to dynamic information can be performed.  
Table 4.9 shows optimizations influenced by a profile information input.

**Table 4.9 Relations Between Profile Information and Optimization**

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

Notes: 1. SHC: C/C++ program for SH

SHA: Assembly program for SH

H8C: C/C++ program for H8

H8A: Assembly program for H8

2. Movement is provided not in the function unit, but in the input file unit.

- Remarks

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

## CAchesize

## Cache Size

Link/Library <Optimize>[Cache size :]

- Command Line Format

CAchesize = <suboption>

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

- Description

Specifies a cache size and cache line size.

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

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

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

- Remarks

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

## SYmbol\_forbid, SAMECode\_forbid, Variable\_forbid, FUnction\_forbid, Absolute\_forbid

## Optimization Partially Disabled

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

- Command Line Format

SYmbol\_forbid = <symbol name> [...]

SAMECode\_forbid = <function name> [...]

Variable\_forbid = <symbol name> [...]

FUnction\_forbid = <function name> [...]

Absolute\_forbid = <address> [+<size>] [...]

- Description

Disables optimization for the specified symbol or address range. Specify an address or the size in the hexadecimal notation. For a C/C++ variable or C function name, add an underscore (\_) at the head of the definition name in the program. For a C++ function, enclose the definition name in the program with double quotation marks including the parameter strings. When the parameter is void, specify as "<function name>()".

Table 4.10 shows the suboptions.

**Table 4.10 Suboptions of Show Option**

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

- Example

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

- Remarks

If **optimize** is not specified, this option is unavailable.



## 4.2.5 Section Options

**Table 4.11 Section Category Options**

Item	Command Line Format	Dialog Menu	Specification
Section address	START = <sub>[...] <sub>: <section name> [ { :   , } <section name> [...]] [/<address>]	Link/Library <Section> [Show entries for :] [Section]	Specifies a section start address
Symbol address file	FSymbol = <section name>[...]	Link/Library <Section> [Show entries for :] [Symbol file]	Outputs externally defined symbol addresses to a definition file.

### START

### Section Address

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

- Command Line Format

START = <suboption> [...]

<suboption>: <section name> [ { : | , } <section name> [...]] [ / <address>]

- Description

Specifies the start address of the section. Specify an address in the hexadecimal notation.

Two or more sections can be allocated to the same address by separating them with a colon (:).

The section name can be specified using wildcards (\*). Sections specified using wildcards are expanded according to the input order.

Sections specified at a single address are allocated in the specification order.

Objects in a single section are allocated in the specification order of the input file or the input library.

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

A section which is not specified with the **start** option is allocated after the last allocation address.

- Example

```
start=P,C,D*/100,R1:R2/8000 ; D1 and D2 are assumed to be in the section starting
                           ; as D.
```

```
ROM=D1=R1,D2=R2
```

Allocates P, C, D1, and D2 to the addresses starting from 0x100 in that order. Both R1 and R2 are allocated to 0x8000.

```
input=a.obj b.obj ; a.obj uses symbols in d.lib and b.obj uses symbols in c.lib.
```

```
library=c.lib,d.lib;
```

```
start=P/100 ; The allocation order in the P section is a(P), b(P), c(P), d(P).
```

- Remarks

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

## FSymbol

## Symbol Address File

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

- Command Line Format  
FSymbol = <section name> [...]
- Description  
Outputs externally defined symbols in the specified section to a file in the assembler directive format.  
The file name is <output file>.fsy.

- Example

```
fSymbol = sct2, sct3
```

```
output=test.abs
```

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

[Output example of **test.fsy**]

```
;OPTIMIZING LINKAGE EDITOR GENERATED FILE 1999.11.26
```

```
;fsymbol = sct2, sct3
```

```
;SECTION NAME = sct2
```

```
.export _f
```

```
_f: .equ h'00000000
```

```
.export _g
```

```
_g: .equ h'00000016
```

```
;SECTION NAME = sct3
```

```
.export _main
```

```
_main: .equ h'00000020
```

```
.end
```

- Remarks

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

## 4.2.6 Verify Options

**Table 4.12 Verify Category Options**

Item	Command Line Format	Dialog Menu	Specification
Address check	CPU = {<cpu information file name>	Link/Library	Specifies a CPU information file.
	{<memory type> = <address range>[,...] <memory type>: { ROm   RAm   XROm   XRAm   YROm   YRAm } <address range>: <start address> -<end address>	<Verify> [CPU information check :]	Specifies a specifiable allocation range for section addresses.

### CPu

### Address Check

Verify[CPU information check:]

- Command Line Format

CPu={<cpu information file name>  
  | {<memory type>} = <address range> [,...]}  
<memory type>: { ROm | RAm | XROm | XRAm | YROm | YRAm }  
<address range>: <start address> - <end address>

- Description

Checks section allocation addresses.

**xrom** and **xram** specify the x memory areas and **yrom** and **yram** specify the y memory areas in the DSP.

Specify an address range in which a section can be allocated in hexadecimal notation. The memory type attribute is used for the inter-module optimization .

The CPU information files created with the CPU information analyzer (cia) attached to a former version product can be specified.

- Example

cpu=ROM=0-FFFF, RAM=10000-1FFFF

Checks that section addresses are allocated within the range from 0 to FFFF or from 10000 to 1FFFF.

Object movement is not provided between different attributes with the inter-module optimization .

- Remarks

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

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

## 4.2.7 Other Options

**Table 4.13 Other Category Options**

Item	Command Line Format	Dialog Menu	Specification
End code	S9	Link/Library <Other> [Miscellaneous options :] [Always output S9 record at the end]	Always outputs the S9 record.
Stack information file	STACK	Link/Library <Other> [Miscellaneous options :] [Stack information output]	Outputs a stack use information file.
Debug information compression	COmpress <u>NOCompress</u>	Link/Library <Other> [Miscellaneous options :] [Compress debug information]	Compresses debug information Does not compress debug information
Memory occupancy reduction	MEMory = [ <u>High</u>   Low ]	Link/Library <Other> [Miscellaneous options :] [Low memory use during linkage]	Specifies the memory occupancy when an input file is loaded
Symbol name modification	REName = <sub>[,...] <sub>: {<file name> (<name>=<name>[,...])   <module name> (<name><name>[,...]) }	Link/Library <Other> [User defined options :]	Modifies a symbol name or section name.
Symbol name deletion	DELeTe = <sub>[,...] <sub>: {<module name>   [ <file name> (<name>[,...]) }	Link/Library <Other> [User defined options :]	Deletes a symbol name or section name.
Module replacement	REPlace = <sub>[,...] <sub>: <file> [ (<module>[,...]) ]	Link/Library <Other> [User defined options :]	Replaces modules of the same name in a library file.
Module extraction	EXTRact = <module>[,...]	Link/Library <Other> [User defined options :]	Extracts the specified module in a library file.
Debug information deletion	STRip	Link/Library <Other> [User defined options:]	Deletes debug information in an absolute file or a library file.
Message level	CHange_message=<sub>[,...] <sub>: {Information   Warning   Error } [=<error number> [-<error number>] [,...]	Link/Library <Other> [User defined options:]	Modifies message levels.

**Table 4. 13 Other Category Options (cont)**

Item	Command Line Format	Dialog Menu	Specification
Local symbol Hide name hide		Link/Library <Other> [User defined options:]	Deletes local symbol name information

## S9 End Code

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

- Command Line Format  
S9
- Description  
Outputs the S9 record at the end even if the entry address exceeds 0x10000.
- Remarks  
When **form=stype** is not specified, this option is unavailable.

## STACK Stack Information File

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

- Command Line Format  
STACK
- Description  
Outputs a stack consumption information file.  
The file name is <output file name>.sni.
- Remarks  
When **form={ object | relocate | library }** or **strip** is specified, this option is unavailable.

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

- Command Line Format  
CCompress  
NOCCompress
- Description  
Specifies whether debug information is compressed.  
When **compress** is specified, the debug information is compressed.  
When **nocompress** is specified, the debug information is not compressed.  
By compressing the debug information, the debugger loading speed is improved. If the **nocompress** option is specified, the link speed is improved.  
If this option is omitted, the default is **nocompress**.
- Remarks  
When **form**=**{ object | relocate | library | hexadecimal | stype | binary }** or **strip** is specified, this option is unavailable.

## MEMory

## Memory Occupancy Reduction

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

- Command Line Format  
MEMory = [ High | Low ]
- Description  
Specifies the memory size occupied for linkage.  
When **memory = high** is specified, the processing is the same as usual.  
When **memory = low** is specified, the linkage editor loads the information necessary for linkage in smaller units to reduce the memory occupancy. This increases file accesses and processing becomes slower when the occupied memory size is less than the available memory capacity.  
**memory = low** is effective when processing is slow because a large project is linked and the memory size occupied by the linkage editor exceeds the available memory in the machine used.
- Remarks  
When one of the following options is specified, this option is unavailable:  
optimize, compress, delete, rename, map, stack, and  
combination of list and show=reference  
Some combinations of this option and the input or output file format are unavailable. For details, refer to table 4.4 of section 4.2.2, Output Options.

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

- Command Line Format

REName = <suboption> [...]

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

- Description

Modifies a symbol name or a section name.

Symbol names or section names in a specific file or library in a module can be modified.

For a C/C++ variable name, add an underscore (\_) at the head of the definition name in the program.

When a function name is modified, the operation is not guaranteed.

If the specified name matches both section and symbol names, the symbol name is modified.

If there are several files or modules of the same name, the priority depends on the input order.

- Example

rename=(\_sym1=data) ; Modifies **sym1** to **data**.

rename=lib1(P=P1) ; Modifies the section **P** to **P1** in the library module **lib1**.

- Remarks

When **extract** or **strip** is specified, this option is unavailable.

## DELete

## Symbol Name Deletion

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

- Command Line Format

DELete = <suboption> [...]

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

- Description

Deletes an external symbol name or library module.

Symbol names or modules in the specified file can be deleted.

For a C/C++ variable name or C function name, add an underscore (\_) at the head of the definition name in the program. For a C++ function name, enclose the definition name in the program with double quotation marks including the parameter strings. If the parameter is void, specify as "<function name>()". If there are several files or modules of the same name, the file that is input first is applied.

When a symbol is deleted using this option, the object is not deleted but the attribute is changed to the internal symbol.

- Example



`delete=(_sym1)` ; Deletes the symbol **\_sym1** in all files.

`delete=file1.obj(_sym2)` ; Deletes the symbol **\_sym2** in the input file **file1.obj**.

- Remarks

When **extract** or **strip** is specified, this option is unavailable.

## REPlace

## Module Replacement

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

- Command Line Format

REPlace = <suboption> [...]

<suboption>: <file name> [ ( <module name> [...]) ] }

- Description

Replaces library modules.

Replaces the specified file or library module with the module of the same name in the library specified with the **library** option.

- Example

`replace=file1.obj` ; Replaces the module **file1** with the module **file1.obj**.

`replace=lib1.lib(md11)` ; Replaces the module **md11** with the module **md11** in the input  
; library file **lib1.lib**.

- Remarks

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

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

- Command Line Format  
EXTract = <module name> [...]
- Description  
Extracts library modules.  
Extract the specified library module from the library file specified using the **library** option.
- Example  
extract=file1 ; Extracts the module **file1**.
- Remarks  
When **form**=**{absolute | hexadecimal | stype | binary}** or **strip** is specified, this option is unavailable.

## STRip

## Debug Information Deletion

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

- Command Line Format  
STRip
- Description  
Deletes debug information in an absolute file or library file.  
When the **strip** option is specified, one input file should correspond to one output file.
- Example  
input=file1.abs file2.abs file3.abs  
strip  
Deletes debug information of **file1.abs**, **file2.abs**, and **file3.abs**, and outputs this information to **file1.abs**, **file2.abs**, and **file3.abs**, respectively. Files before debug information is deleted are backed up in **file1.abk**, **file2.abk**, and **file3.abk**.
- Remarks  
When **form**=**{object | relocate | hexadecimal | stype | binary}** is specified, this option is unavailable.

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

- Command Line Format

CHange\_message = <suboption> [,...]

<suboption>: <error level> [= <error number> [-<error number>] [,...]]

<error level>: {Information | Warning | Error}

- Description

Modifies the level of information, warning, and error messages.

Specifies the execution continuation or abort at the message output.

- Example

change\_message=warning=2310

Modifies L2310 to the warning level and specifies execution continuation at L2310 output.

change\_message=error

Modifies all information and warning messages to error level messages.

When a message is output, the execution is aborted.

unavailable.

## Hide

## Local Symbol Name Hide

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

- Command Line Format

Hide

- Description

Deletes local symbol name information from the output file. Since all the name information regarding local symbols is deleted, local symbol names cannot be checked even if the file is opened with a binary editor. This option does not affect the operation of the generated file.

Use this option to keep the local symbol names secret.

The following types of symbol names are hidden:

C source: Variable or function names specified with the **static** qualifiers

C source: Label names for the **goto** statements

Assembly source: Symbol names of which external definition (reference) symbols are not declared

- Example

The following is a C source example in which this option is valid:

```
int g1;
int g2=1;
const int g3=3;

static int s1;          //<- The static variable name will be hidden.
static int s2=1;        //<- The static variable name will be hidden.
static const int s3=2;  //<- The static variable name will be hidden.

static int sub1()        //<- The static function name will be hidden.
{
    static int s1;       //<- The static variable name will be hidden.
    int l1;

    s1 = l1; l1 = s1;
    return(l1);
}

int main()
{
    sub1();
    if (g1==1)
        goto L1;
    g2=2;
L1:          //<- The label name of the goto statement
            // will be hidden.

    Return(0);
}
```

- Remarks

This option is available only when the output file format is specified as **absolute**, **relocate**, or **library**.

When the input file was compiled or assembled with the **goptimize** option specified, this option is unavailable if the output file format is specified as **relocate** or **library**.

To use this option with optimization by the **map** option, do not use this option for the first linkage, and use it only for the second linkage.

The symbol names in the debug information are not deleted by this option.

4.2.8 Subcommand File Option

Table 4.14 Subcommand Tab Option

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

Subcommand

Subcommand File

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

- Command Line Format  
SUBcommand = <file name>
- Description  
Specifies options with a subcommand file.  
The format of the subcommand file is as follows:  
<option> { = | Δ } [<suboption> [...] ] [ Δ& ] [;<comment>]  
The option and suboption are separated by an “=” sign or a space.  
For the **input** option, suboptions are separated by a space.  
One option is specified per line in the subcommand file.  
If a subcommand description exceeds one line, the description can be allowed to overflow to the next line by using an ampersand (&).  
The **subcommand** option cannot be specified in the subcommand file.
- Example  
Command line specification: optlnk file1.obj -sub=test.sub file4.obj  
Subcommand specification: input file2.obj file3.obj ; This is a comment.  
                                  library lib1.lib, & ; Specifies line continued.  
                                  lib2.lib  
Option contents specified with a subcommand file are expanded to the location at which the subcommand is specified on the command line and are executed.  
The order of file input is **file1.obj**, **file2.obj**, **file3.obj**, and **file4.obj**.

## 4.2.9 CPU Option

**Table 4.15 CPU Tab Option**

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

### SBr

### SBR Address Specification

- Command Line Format  
SBr = { <address> | User }
- Description  
Specifies the SBR address.  
When the SBR address is specified in this option, optimization using the abs8 area is available.  
When **user** is specified in this option, optimization for the abs8 area is disabled.
- Remarks  
This option is available only when the CPU is H8SX.  
If more than one SBR address is specified within the source or by tool options, the optimizing linkage editor assumes that **user** is specified regardless of this option setting.

## 4.2.10 Options Other Than Above

**Table 4.16 Options Other Than Above**

Item	Command Line Format	Dialog Menu	Specification
Copyright	<u>LOgo</u> NOLOgo	-	Output Not output
Continuation	END	-	Executes option strings already input, inputs continuing option strings and continues processing.
Termination	EXIt	-	Specifies the termination of option input.
Notification of unreferenced defined symbol	MSg_unused	-	Notifies the user of the defined symbol which is never referenced

### LOgo, NOLOgo

### Copyright

None (nologo is always available.)

- Command Line Format

LOgo

NOLOgo

- Description

Specifies whether the copyright is output.

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

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

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

None

- Command Line Format

END

- Description

Executes option strings specified before END. After the linkage processing is terminated, option strings that are specified after END are input and the linkage processing is continued. This option cannot be specified on the command line.

- Example

```
input=a.obj,b.obj           ; processing (1)
start=P,C,D/100,B/8000      ; processing (2)
output=a.abs                ; processing (3)
end
input=a.abs                  ; processing (4)
form=stype                   ; processing (5)
output=a.mot                 ; processing (6)
```

Executes the processing from (1) to (3) and outputs **a.abs**. Then executes the processing from (4) to (6) and outputs **a.mot**.

## EXIt

## Termination Processing

None

- Command Line Format

EXIt

- Description

Specifies the end of the option specifications.

This option cannot be specified on the command line.

- Example

Command line specification: `optlnk -sub=test.sub -nodebug`

```
test.sub:      input=a.obj,b.obj           ; processing (1)
                start=P,C,D/100,B/8000      ; processing (2)
                output=a.abs                ; processing (3)
                exit
```

Executes the processing from (1) to (3) and outputs **a.abs**.

The **nodebug** option specified on the command line after **exit** is executed is ignored.



# Section 5 Standard Library Generator Operating Method

## 5.1 Comand Line Format

The format of the command line is as follows:

```
lbg38 [ $\Delta$ <option string>...]  
      <option string>:-<option>[=<suboption>[,...]]
```

## 5.2 Option Descriptions

Options and suboptions of the standard library generator are based on the C/C++ compiler options. The following section describes the difference between the options and suboptions of the standard library generator and those of the C/C++ compiler. For details on C/C++ compiler options, refer to section 2, C/C++ Compiler Operating Method.

In the command line format, uppercase letters indicate abbreviations. The format of the dialog menus that correspond to the HEW is as follows:

Tab name <Category>[Item] ...

## 5.2.1 Additional Options

Table 5.1 shows additional options.

**Table 5.1 Additional Options**

Item	Command Line Format	Dialog Menu	Specification
Header file	Head = <sub>[,...] <sub>:{ <u>ALL</u> RUNTIME CTYPE MATH MATHF STDARG STDIO STDLIB STRING IOS NEW COMPLEX CPPSTRING }	Standard Library <Standard Library> [Category :]	Specifies parts to be generated All library functions Runtime routine ctype.h + runtime routine math.h + runtime routine mathf.h + runtime routine stdarg.h + runtime routine stdio.h + runtime routine stdlib.h + runtime routine string.h + runtime routine ios + runtime routine new + runtime routine complex + runtime routine string + runtime routine
Output file	OUTPut = <file name>	Standard Library <Object> [Output file path :]	Specifies an output library file name
Reentrant library	REent	Standard Library <Object> [Generate reentrant library]	Creates reentrant library

## Head

Standard Library <Standard Library>[Category :]

- Command Line Format

```
Head = <sub>[,...]  
<sub>:{ ALL |  
        RUNTIME |  
        CTYPE |  
        MATH |  
        MATHF |  
        STDARG |  
        STDIO |  
        STDLIB |  
        STRING |  
        IOS |  
        NEW |  
        COMPLEX |  
        CPPSTRING }
```

- Description

Specifies one or more categories to be generated with a header file name.

For relationships between header files and library functions, refer to section 10.3, C/C++ Libraries. The runtime routine is always generated.

The default interpretation of this option is **head=all**.

- Example

```
lb38 -output=h8s.lib -head=mathf -cpu=2600a
```

Compiles library functions defined by mathf.h and runtime routine with option: -cpu=2600a, and outputs library file h8s.lib.

## OUTPut

Standard Library <Object>[Output file path :]

- Command Line Format

OUTPut = <File name>

- Description

Specifies an output file name. The default of this option is **output=stdlib.lib**.

- Example

```
lb38 -output=h8s.lib -optimize -speed -goptimize -cpu=2600a
```

Compiles all standard library source files with options: -optimize -speed -goptimize -cpu=2600a, and outputs library file h8s.lib.

## REent

Standard Library <Object> [Generate reentrant library]

- Command Line Format

REent

- Description

Creates reentrant functions. Note that the **rand** and **srand** functions are not reentrant functions. Also note that the behavior of subsequent calls of the **strtok** function using the same string is not guaranteed.

- Example (user program)

```
#define _REENTRANT
#include <stdlib.h>
```

- Remarks

When reentrant functions are linked, use #define statements to define macro names (**#define \_REENTRANT**) or use the **define** option to define **\_REENTRANT** at compilation before including standard include files in the program.

## 5.2.2 Options Unavailable for Standard Library Generator

Table 5.2 shows C/C++ compiler options that cannot be specified for the standard library generator. If any of the options listed in table 5.2 are specified, these specifications are ignored.

**Table 5.2 Options Not Unavailable for Standard Library Generator**

Item	Option	Compiler Interpretation	Description
Include file directory	Include	N/A	—
Macro name definition	DEFine	N/A	—
Disable preprocessor #line output	NOLINE	N/A	—
Message output control	Message NOMessage	NOMessage	No output
Preprocessor inline output	PREProcessor	N/A	—
Object type	Code	Code = Machinecode	Outputs machine code program
Debugging information	DEBug NODEBug	NODEBug	No output
Object file output	Object NOObject	Object	Output
Template instance generation	Template	N/A	No template function used
Listing file	List NOList	NOList	No output
Listing format	SHow	N/A	—
Comment nesting	COMment	N/A	No comment nesting function used
MAC register	MAcsave	N/A	No interrupt function included
Message level	CHAnge_message	N/A	—
Selecting C or C++ language	LANG	N/A	Determined by an extension
Disable of Copyright output	LOGo NOLOGo	NOLOGo	Copyright output disabled
Character code select in EUc string literals	Sjjs LATin1	N/A	No character code used
Japanese character conversion within object code	OUtcode	N/A	No character code used

### 5.2.3 Notes on Specifying Options

When options are specified, follow the rules below:

- (1) Specify the same options as in compiling for options **cpu**, **regparam**, **structreg/nostructreg**, **longreg/nolongreg**, **stack**, **double=float**, **byteenum**, **pack**, **rtti=on/off**, **exception/noexception**, **bit\_order=left/right**, **indirect=normal/extended**, **ptr16**, and **sbr**.
- (2) In order to use `#pragma global_register`, specify a header file that consists of the `#pragma global_register` declaration with the **preinclude** option. When the HEW is used, specify it with Standard Library <Other>[User defined options :].

## Section 6 Operating Stack Analysis Tool

### 6.1 Overview

The stack analysis tool displays the stack amount by reading the stack information file (\*.sni) output by the optimizing linkage editor or the profile information file (\*.pro) output by the simulator debugger.

For the stack amount of the assembly program (assembled by the assembler) that cannot be output in the stack information file, the information can be added or modified by using the edit function. In addition, the assembler Ver.6.01 can output the stack size for symbol and the stack amount of whole systems can be calculated.

The information on the edited stack amount can be saved and read as the call information file (\*.cal).

### 6.2 Starting the Stack Analysis Tool

To start the stack analysis tool, select [Run...] from the start menu of Windows® and specify Call.exe for execution.

When the HEW is used, select [Program] from the start menu of Windows®, select the HEW menu, and then select Call Walker.

After the HEW is started, the stack analysis tool can also be started from the [Tools] menu.

For details on operation, refer to the help of the stack analysis tool.





## Section 7 Environment Variables

### 7.1 Environment Variables List

The environment variables to be used by the compiler are listed in table 7.1.

**Table 7.1 Environment Variables**

Environment Variable	Description
path	<p>Specifies a storage directory for the execution file.</p> <p>Specification format:</p> <p>PC version: C&gt; path = &lt;execution file path name&gt;[;&lt;previous path name&gt;;...]</p> <p>UNIX C shell: %set path = (&lt;execution file path name&gt; \$path)</p> <p>UNIX Bourne shell: %PATH = :&lt;execution file path name&gt; [;&lt;previous path name&gt;;...]</p> <p>%export PATH</p>

**Table 7.1 Environment Variables (cont)**

**Environment**

**Variable Description**

H38CPU	Specifies the CPU type overridden by the compiler or assembler <b>cpu</b> option.		
<CPU/operating mode>	Bit Width in Address Space <value1>	Multiplier and Divider Specification <value2>	
<b>AE5</b>	—	—	
<b>H8SXN</b>	—	M   D   MD	
<b>H8SXM</b>	20   24 (24)	M   D   MD	
<b>H8SXA</b>	20   24   28   32 (24)	M   D   MD	
<b>H8SXX</b>	28   32 (32)	M   D   MD	
<b>2600n</b>	—	—	
<b>2600a</b>	20   24   28   32 (24)	—	
<b>2000n</b>	—	—	
<b>2000a</b>	20   24   28   32 (24)	—	
<b>300hn</b>	—	—	
<b>300ha</b>	20   24 (24)	—	
<b>300</b>	—	—	
<b>300l</b>	—	—	

The default value is enclosed by parentheses, ( ).

When the specification of CPU by H38CPU environment variable and the **cpu** option differs, a warning message is displayed. **Cpu** option has priority over H38CPU specification.

Specification format:

PC version: C> set H38CPU = <CPU/operating mode>[:<value1>][:<value2>]

UNIX C shell: % setenv H38CPU = <CPU/operating mode>[:<value1>][:<value2>]

UNIX Bourne shell: % H38CPU = <CPU/operating mode>[:<value1>][:<value2>]  
% export H38CPU

**Table 7.1 Environment Variables (cont)**

<b>Environment Variable</b>	<b>Description</b>
CH38 *	<p>Specifies an include file storage directory</p> <p>The search order for system include files is any directory specified by an <b>include</b> option, then this directory.</p> <p>The search order for user include files is the current directory, any directory specified by an <b>include</b> option, then this directory.</p> <p>If environment variable CH38 is not specified, /usr/CH38 is assumed in the UNIX version. The PC version does not have default.</p> <p>Specification format:</p> <p>PC version: C&gt; set CH38 = &lt;include path name&gt; [;&lt;include path name&gt;;...]</p> <p>UNIX C shell: % setenv CH38 = &lt;include path name&gt;[:&lt;include path name&gt;:...]</p> <p>UNIX Bourne shell: % CH38 = &lt;include path name&gt;[:&lt;include path name&gt;:...] % export CH38</p>
CH38TMP	<p>Specifies a directory in which the compiler creates temporary files. If CH38TMP is not specified, temporary files are created in the current directory.</p> <p>Specification format:</p> <p>PC version: C&gt; set CH38TMP = &lt;temporary file path name&gt;</p> <p>UNIX C shell: % setenv CH38TMP = &lt;temporary file path name&gt;</p> <p>UNIX Bourne shell: % CH38TMP = &lt;temporary file path name&gt; % export CH38TMP</p>
CH38SBR	<p>Specifies a short address base register (SBR) for the compiler. The method of specification is the same as that of the compiler's <b>sbr</b> option.</p> <p>Specification format:</p> <p>PC version: C&gt; set CH38SBR = &lt;address&gt;</p> <p>UNIX C shell: % setenv CH38SBR = &lt;address&gt;</p> <p>UNIX Bourne shell: % CH38SBR = &lt;address&gt; % export CH38SBR</p>

**Table 7.1 Environment Variables (cont)**

<b>Environment Variable</b>	<b>Description</b>
HLNK_LIBRARY1 HLNK_LIBRARY2 HLNK_LIBRARY3	<p>Specifies a default library name for the optimizing linkage editor. Libraries which are specified by a <b>library</b> option are linked first. Then, if there is an unresolved symbol, the default libraries are searched in the order 1, 2, 3.</p> <p>Specification format:</p> <p>PC version: C&gt; set HLNK_LIBRARY1 = &lt;library name 1&gt; C&gt; set HLNK_LIBRARY2 = &lt;library name 2&gt; C&gt; set HLNK_LIBRARY3 = &lt;library name 3&gt;</p> <p>UNIX C shell: % setenv HLNK_LIBRARY1 = &lt; library name 1&gt; % setenv HLNK_LIBRARY2 = &lt; library name 2&gt; % setenv HLNK_LIBRARY3 = &lt; library name 3&gt;</p> <p>UNIX Bourne shell: % HLNK_LIBRARY1 = &lt; library name 1&gt; % export HLNK_LIBRARY1 % HLNK_LIBRARY2 = &lt; library name 2&gt; % export HLNK_LIBRARY2 % HLNK_LIBRARY3 = &lt; library name 3&gt; % export HLNK_LIBRARY3</p>
HLNK_TMP	<p>Specifies a directory in which the optimizing linkage editor creates temporary files. If HLNK_TMP is not specified, temporary files are created in the current directory.</p> <p>Specification format:</p> <p>PC version: C&gt; set HLNK_TMP = &lt;temporary file path name&gt;</p> <p>UNIX C shell: % setenv HLNK_TMP = &lt;temporary file path name&gt;</p> <p>UNIX Bourne shell: % HLNK_TMP = &lt;temporary file path name&gt; % export HLNK_TMP</p>
HLNK_DIR *	<p>Specifies an input file storage directory for the optimizing linkage editor. The search order for files which are specified by the <b>input</b> and the <b>library</b> options is the current directory then this directory.</p> <p>However, when a wildcard is used in the file specification, only the current directory is searched.</p> <p>Specification format:</p> <p>PC version: C&gt; set HLNK_DIR = &lt;input file path name&gt; [&lt;input file path name &gt;;...]</p> <p>UNIX C shell: % setenv HLNK_DIR = &lt;input file path name&gt;[:&lt;input file path name &gt;:...]</p> <p>UNIX Bourne shell: % HLNK_DIR = &lt;input file path name&gt;[:&lt;include path name&gt;:...] % export HLNK_DIR</p>

**Note:** More than one directory can be specified by dividing directories by a semicolon (;) in the PC version, or by a colon (:) in the UNIX version.

## 7.2 Compiler Implicit Declaration

The compiler implicitly defines the macro names according to its version and options specified.

**Table 7.2 Compiler Implicit Declaration**

Option	Implicit Declaration
cpu = 300L	#define __300L__
cpu = 300	#define __300__
cpu = 300HN	#define __300HN__
cpu = 300HA	#define __300HA__
cpu = 2000N	#define __2000N__
cpu = 2000A	#define __2000A__
cpu = 2600N	#define __2600N__
cpu = 2600A	#define __2600A__
cpu = H8SXN	#define __H8SXN__
cpu = H8SXM	#define __H8SXM__
cpu = H8SXA	#define __H8SXA__
cpu = H8SXX	#define __H8SXX__
cpu = <H8SX>:M or MD	#define __HAS_MULTIPLIER__
cpu = <H8SX>:D or MD	#define __HAS_DIVIDER__
cpu = AE5	#define __AE5__
double = float	#define __FLT__
byteenum	#define __BENM__
cpuexpand	#define __CPUEX__
library=intrinsic	#define __INTRINSIC_LIB__
abs16	#define __ABS16__
—	#define __ADDRESS_SPACE__ <sup>*1</sup> *4
—	#define __DATA_ADDRESS_SIZE__ <sup>*2</sup> *4
—	#define __H8__ <sup>*4</sup>
—	#define __RENESAS_VERSION__ <sup>*3</sup> *4
—	#define __HITACHI_VERSION__ <sup>*3</sup> *4
—	#define __RENESAS__ <sup>*4</sup>
—	#define __HITACHI__ <sup>*4</sup>

- Notes:
1. Address width (16, 20, 24, 28, or 32 bits) is defined.
  2. `__DATA_ADDRESS_SIZE__` is defined as 2 or 4 as shown below.
    - 2: 300, normal or middle mode, or advanced or maximum mode with the ptr16 option
    - 4: Advanced or maximum mode without the ptr16 option
  3. The value of `__RENESAS_VERSION__` and `__HITACHI_VERSION__` is as follows:  
C source program: `__RENESAS_VERSION__==0xaabb`  
aa: version  
bb: revision  
Example definition in the compiler:  
`#define __RENESAS_VERSION__ 0x0301 //Version 3.1C`  
`#define __RENESAS_VERSION__ 0x0400 //Version 4.0`
  4. Always defined.

## Section 8 File Specifications

### 8.1 Naming Files

A standard file extension is automatically added to the name of a compiled file when the file extension is omitted at file-naming. The standard file extensions used in the development environment are shown in table 8.1.

**Table 8.1 Standard File Extensions Used in the Development Environment**

<b>No.</b>	<b>File Extension</b>	<b>Description</b>
1	c	Source program file written in C
2	cpp, cc, cp	Source program file written in C++
3	h	Include file
4	lis, lst * <sup>1</sup>	C source program listing file
5	lis, lpp * <sup>1</sup>	C++ source program listing file
6	p	File after the expansion by the C source program preprocessor
7	pp	File after the expansion by the C++ source program preprocessor
8	src, mar	Assembly source program file
9	exp	File after the expansion by the assembly source program preprocessor
10	lis	Assembly source program listing file
11	obj	Relocatable object program file
12	rel	Relocatable load module file
13	abs	Absolute load module file
14	map	Linkage map listing file
15	lib	Library file
16	lbp	Library listing file
17	mot	S-type format
18	hex	HEX format
19	bin	Binary file
20	fsy	Symbol address file for optimizing linkage editor output
21	sni	Stack information file
22	pro	Profile information file
23	dbg	DWARF2-format debugging information file
24	rti	Object that includes a definition specified in the file with extension td
25	cal	Calling information file

Note: 1. The extension is “lis” for the UNIX version, and “lst” or “lpp” for the PC version.

Do not name a file a name beginning with “rti\_”, which indicates a file reserved for system use.

Table 8.2 lists the extensions for files that are output under the tpldir folder generated by each project.



**Table 8.2 tpldir Folder Output File**

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

For general rules on naming files, refer to the user's manual of the host computer because naming rules vary according to each host computer.

## 8.2 Compiler Listings

This section deals with compiler listings and their formats.

### 8.2.1 Structure of Compiler Listings

Table 8.3 shows the structure and contents of compiler listings.

**Table 8.3 Structure and Contents of Compiler Listings**

List Structure	Contents	Option Specification Method	Default
Source listing information	Source program listing* <sup>1</sup>	show=source show=nosource	Output
	Source program listing of include file and after macro expansion* <sup>2</sup>	show=expansion show=noexpansion	No output
Error information	Errors detected during compilation	—	Output
Symbol allocation information	Variables allocated to stack frame of a function	show=allocation show=noallocation	No output
Object information	Machine code in object program and the assembly code	show=object show=noobject	No output
Statistics information	Length of each section (byte), number of symbols, and object types	show=statistics show=nostatistics	Output

- Notes: 1. Source program listings are inserted in the object information when the **noexpansion** and **object** suboptions are specified simultaneously.
2. The source program listing of include files and after macro expansion is valid only when **show=source** is specified.

## 8.2.2 Source Listing

The source listing may be output in two ways. When **show=noexpansion** is specified, the unpreprocessed source program listing is output. When **show=expansion** is specified, the preprocessed source program listing is output. Figures 8.1 (a) and (b) show these output formats, respectively. In addition, figure 8.1 (b) shows the differences between them with bold characters.

```
***** SOURCE LISTING *****

      Line Pi 0-----1-----2-----3-----4-----5-----6-----((
FILE NAME: m0260.c
      1 [1] #include "header.h"
      2
      3     int sum2(void)
      4     {   int j;
      5
      6     #ifdef SMALL
      7         j=SML_INT;
      8     #else
      9         j=LRG_INT;
     10     #endif
     11
     12         return j; /*
continue 123456789012345678901234567890123456789012345678901234567890((
23456789012345678901234567890 */
      13     }
      [2]
```

Figure 8.1 (a) Source Listing Output for show=noexpansion

\*\*\*\*\* SOURCE LISTING \*\*\*\*\*

```

Line Pi 0-----1-----2-----3-----4-----5-----6--((
FILE NAME: m0260.c
1 [1] #include "header.h"
FILE NAME: header.h
1      #define SML_INT          1
2      #define LRG_INT          100
FILE NAME: m0260.c
2
3      int sum2(void)
4      {   int j;
5
6      #ifdef SMALL
7 X          j=SML_INT;
8[3] #else
9 E          j=100;
10 [4] #endif
11
12          return j; /* continue123456789012345678901234567890123456789
23456789012345678901234567890 */
13      }
[2]

```

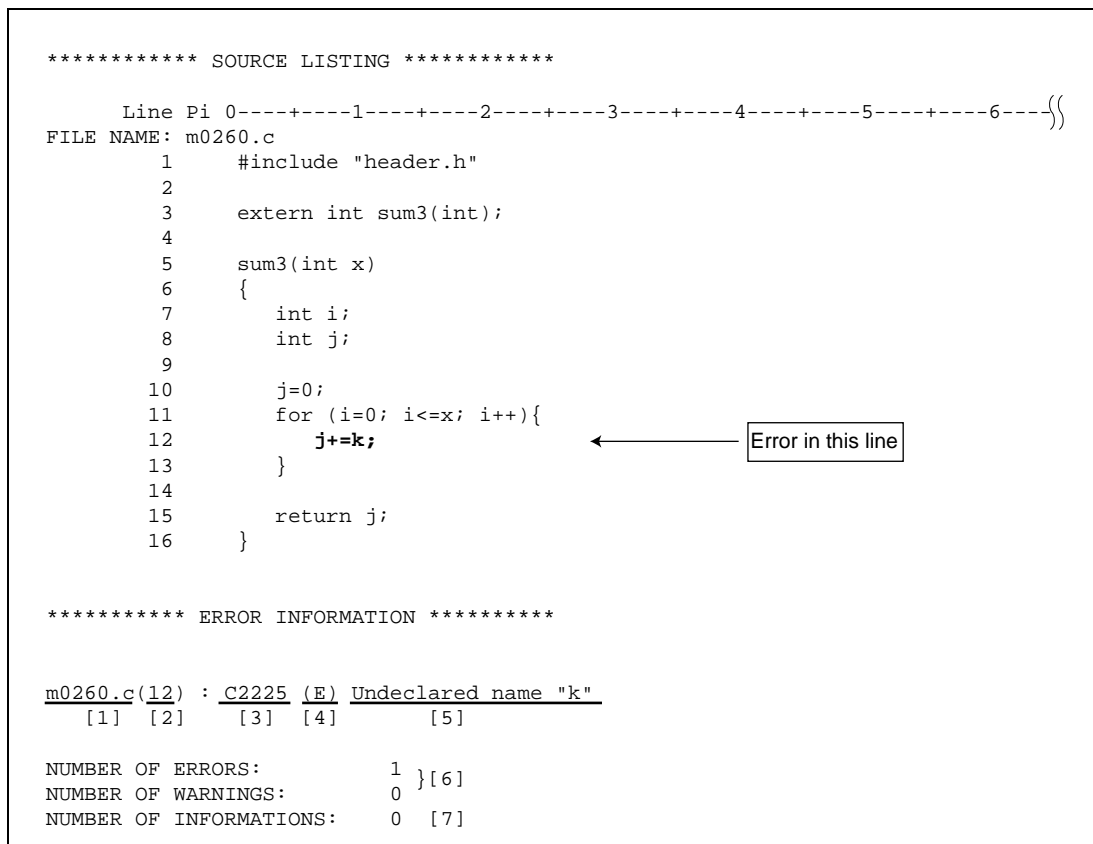
**Figure 8.1 (b) Source Listing Output for show=expansion**

### Description

- [1] Source program file name or include file name
- [2] Line number in source program or include file
- [3] If **show=expansion** is specified and conditional directives such as **#ifdef** and **#elif** are used, a source program line that is not to be compiled is marked with an X.
- [4] If **show=expansion** is specified and **#define** directives are used to expand macros, a line containing a macro expansion is marked with an E.

## 8.2.3 Error Information

Figure 8.2 shows an example of error information.



**Figure 8.2 Source Listing Including Errors and Error Information**

### Description

- [1] The name of the source program in which the error occurred is indicated within the first ten characters.
  - [2] The line number containing the error is shown.
  - [3] The error number identifies the error message.
  - [4] (I) Information level  
(W) Warning level  
(E) Error level  
(F) Fatal level
  - [5] Contents of the error message.
  - [6] The total number of error-level messages and the total number of warning-level messages.
- 160

[7] The total number of information-level messages (only when the **message** option is specified).

## 8.2.4 Symbol Allocation Information

Symbol allocation information is the information of function parameters and local variables. Figure 8.3 shows an example of symbol allocation information when a program is compiled in H8S/2600 advanced mode.

\*\*\*\*\* SOURCE LISTING \*\*\*\*\*

```
Line Pi 0-----1-----2-----3-----4-----5-----6-{{
FILE NAME: m0280.c
1      extern int h(char, char *, double );
2
3      int
4      h(char a, register char *b, double c)
5      {
6          char      *d;
7
8          d= &a;
9          h(*d,b,c);
10         {
11             register int i;
12
13             i= *d;
14             return i;
15         }
16     }
```

\*\*\*\*\* STACK FRAME INFORMATION \*\*\*\*\*

FILE NAME: m0280.c

Function (File m0280.c , Line 4): h

[1]

Parameter Allocation

a		0xffffffff7 saved from R0L	} [2]
b	REG ER5	saved from ER1	
c		0x00000008	

Level 1 (File m0280.c , Line 5)	Automatic/Register Variable Allocation	} [3]
d	0xffffffff2	

Level 2 (File m0280.c , Line 10)	Automatic/Register Variable Allocation	} [3]
i	REG R4	

Parameter Area Size	: 0x00000008 Byte(s)	} [4]
Linkage Area Size	: 0x00000008 Byte(s)	
Local Variable Size	: 0x00000006 Byte(s)	
Temporary Size	: 0x00000000 Byte(s)	
Register Save Area Size	: 0x00000008 Byte(s)	
Total Frame Size	: 0x0000001e Byte(s)	

Figure 8.3 Symbol Allocation Information (cpu=2600a)

## Description

[1] File name in which the function is defined, line number, and function name

[2] Parameter allocation

X saved from Y:	A parameter passed with Y is copied to X at the entry of the function.
REG ERx:	If a parameter is allocated to a register, REG is indicated.
0xfffffx,0x000000xx:	If a parameter is allocated to a stack, the offset from the address by the frame pointer (ER6) is indicated.

[3] Local variable allocation information

This indicates where the local variables declared in a compound statement are stored. If they are allocated to stacks, the offset from the address indicated by ER6 is shown. If they are allocated to registers, REG is displayed.

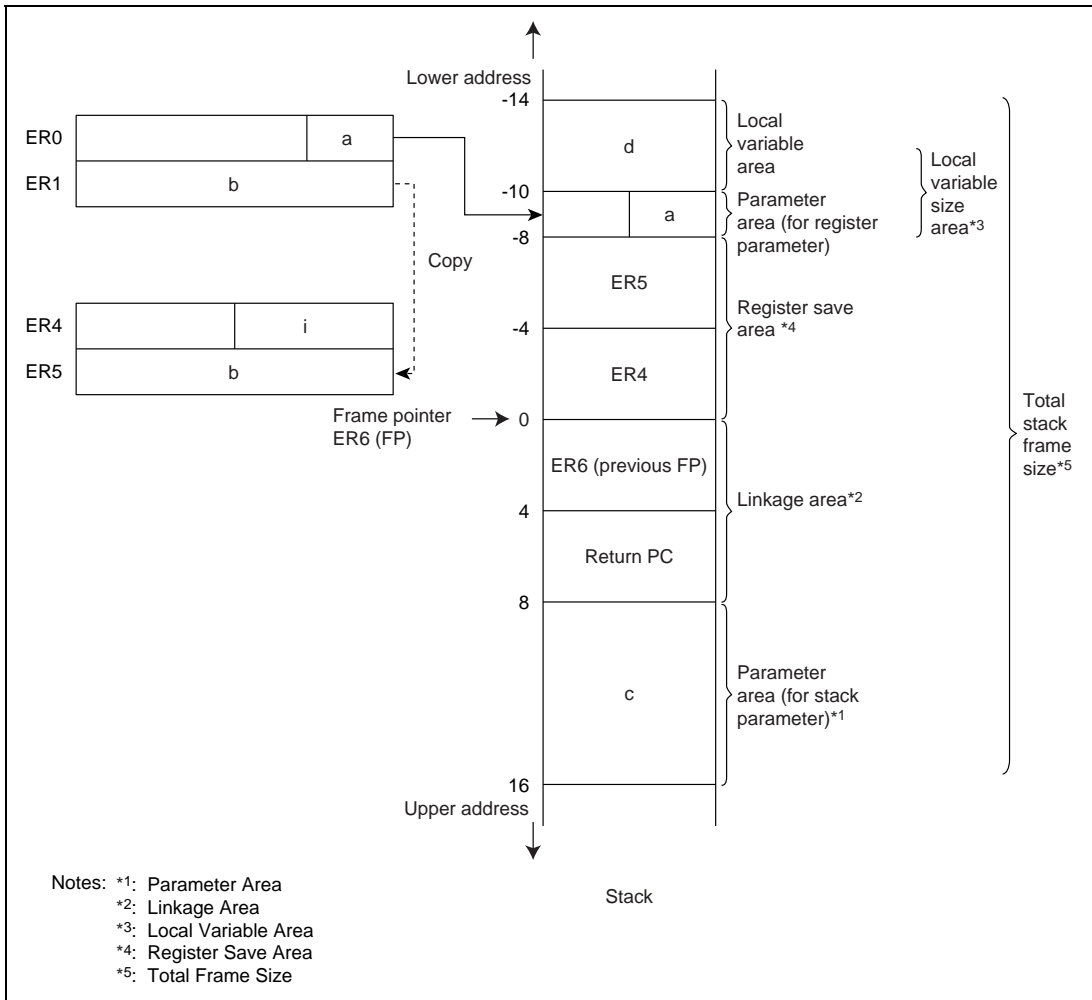
[4] Allocation information on the stack frame used in a function

Parameter Area Size:	The total size of the bath area for parameters allocated to the stack and the area for return value address.
Linkage Area Size:	The total size of the linkage area (return PC area and frame pointer save area, frame pointer save area may not exist) For the interrupt function the size of saving area for CCR and EXR is added, where EXR is only for H8SX, H8S/2600 or H8S/2000.
Local Variable Size:	The total size of both the local variable area in the function and the parameter save area which is reserved when a parameter passed in a register is allocated to the stack.
Temporary Size:	The size of the temporary area used by the compiler in the function.
Register Save Area Size:	The size of the amount of memory required to save the register contents used by the function.
Total Frame Size:	The total size of stack frames allocated in the function.

Note: The following message is output instead of parameter allocation information and local variable allocation information when the option **optimize=1** is specified or when the CPU is H8SX.

Optimize Option Specified : No Allocation Information Available

Figure 8.4 shows an example of stack allocation corresponding to the symbol allocation information shown in figure 8.3.



**Figure 8.4 Stack Allocation Example (cpu=2600a)**

## 8.2.5 Object Information

Figures 8.5 and 8.6 show object listing examples when the source program listing is output to the object information and when not output, respectively.

***** OBJECT LISTING *****						
FILE NAME: m0251.c						
SCT	OFFSET	CODE	LABEL	INSTRUCTION	OPERAND	COMMENT
[1]	[2]	[3]		[4]		
P						; section
	1:	extern int sum(int);				
	2:	[5]				
	3:	int				
	4:	sum(int x)				
00000000			_sum:			; function: sum
	5:	{				
	6:	int i;				
	7:	int j;				
	8:					
	9:	j=0;				
	10:					
	11:	for(i=0; i<=x; i++){				
00000000	1988			SUB.W	E0,E0	
00000002	4000			BRA	L8:8	
00000004			L7:			
00000004	0B58			INC.W	#1,E0	
00000006			L8:			
00000006	1D08			CMP.W	R0,E0	
00000008	4F00			BLE	L7:8	
	12:	j+=1;				
	13:	}				
	14:					
	15:	return;				
	16:	}				
0000000A	5470			RTS		

**Figure 8.5 Object Information When Source Program Listing Is Output (show=source, object, cpu=2600a)**

### Description

- (1) Section name (P, C, D, B) of each section
- (2) The offset indicates the offset address relative to the beginning of each section
- (3) Contents of the offset address of each section
- (4) Assembly code corresponding to machine code
- (5) Line number and contents of source program

Note: When the **show=expansion** option is specified, the object listing is always output in the format shown in figure 8.6.



\*\*\*\*\* OBJECT LISTING \*\*\*\*\*

FILE NAME: m0251.c

<u>SCT</u>	<u>OFFSET</u>	<u>CODE</u>	<u>C LABEL</u>	<u>INSTRUCTION</u>	<u>OPERAND</u>	<u>COMMENT</u>
[1]	[2]	[3]			[4]	
P						; section
	00000000		;*** File m0251.c	, Line 4		; block
			_sum:			; function: sum
			;*** File m0251.c	, Line 5		; block
	00000000	1911	;*** File m0251.c	, Line 9		; expression statement
			SUB.W	R1,R1		
			;*** File m0251.c	, Line 10		; expression statement
	00000002	1988	SUB.W	E0,E0		
			;*** File m0251.c	, Line 10		; for
	00000004	4004	BRA	L8:8		
	00000006		L7:			
			;*** File m0251.c	, Line 10		; block
			;*** File m0251.c	, Line 11		; expression statement
	00000006	0981	ADD.W	E0,R1		
			;*** File m0251.c	, Line 10		; expression statement
	00000008	0B58	INC.W	#1,E0		
	0000000A		L8:			
	0000000A	1D08	CMP.W	R0,E0		
	0000000C	4FF8	BLE	L7:8		
			;*** File m0251.c	, Line 13		; return
	0000000E	0D10	MOV.W	R1,R0		
			;*** File m0251.c	, Line 14		; block
	00000010	5470	RTS			

**Figure 8.6 Object Information When Source Program Listing Is Not Output  
(show=nosource, object, cpu=2600a)**

### Description

- (1) Section name (P, C, D, B) of each section
- (2) The offset indicates the offset address relative to the beginning of each section
- (3) Contents of the offset address of each section
- (4) Assembly code corresponding to machine code

## 8.2.6 Statistics Information

Figure 8.7 shows an example of statistics information.

```
***** SECTION SIZE INFORMATION *****

PROGRAM  SECTION(P):                0x00000012 Byte(s)
CONSTANT SECTION(C):                0x00000000 Byte(s)
DATA     SECTION(D):                0x00000000 Byte(s)
BSS      SECTION(B):                0x00000000 Byte(s)

TOTAL PROGRAM SECTION: 0x00000012 Byte(s) [1]
TOTAL CONSTANT SECTION: 0x00000000 Byte(s)
TOTAL DATA   SECTION: 0x00000000 Byte(s)
TOTAL BSS     SECTION: 0x00000000 Byte(s)

TOTAL PROGRAM SIZE: 0x00000012 Byte(s)

** ASSEMBLER/LINKAGE EDITOR LIMITS INFORMATION **

NUMBER OF EXTERNAL REFERENCE SYMBOLS:    0
NUMBER OF EXTERNAL DEFINITION SYMBOLS:   1 } [2]
NUMBER OF INTERNAL/EXTERNAL SYMBOLS:     3

**** COMPILE CONDITION INFORMATION ****

COMMAND LINE: -sh=allocation -opt=0 test.c [3]
cpu          : 2600a                        [4]
```

**Figure 8.7 Statistics Information**

### Description

- (1) Size of each section and total size of sections
- (2) Number of external reference symbols, number of external definition symbols, and total number of internal and external labels in object program
- (3) Contents of command line specification
- (4) CPU/operating mode

**Note:** Statistics information is not output if an error-level error or fatal-level error has occurred or when option **noobject** is specified. In addition, SECTION SIZE INFORMATION is not output when option **code=asmcode** is specified.

## 8.3 Assembler Listings

### 8.3.1 Structure of Assembler Listings

Table 8.4 shows the structure and contents of assembler listings.

**Table 8.4 Structure and Contents of Assembler Listings**

List Structure	Contents	Option Specification Method	Default
Source listing information	Shows information relating to source program	source	Output
Cross reference listing information	Shows information relating to source program symbols	cross_reference	Output
Section listing information	Shows information relating to source program section	section	Output

Note: All of the options listed are valid when the **list** option is specified.

### 8.3.2 Source Listing

Source listing information is shown. Figure 8.8 shows an example of source listing.

1	1	.CPU 2600A:32
2	2	;
3 00000000	3	.SECTION AAA, CODE, ALIGN=2
4 00000000	4	START
5 00000000 7A0700000000	5	MOV.L #STACK:32, SP
6 00000006 F800	6	MOV.B #0:8, R0L
7 00000008 6AA800000000	7	MOV.B R0L, @ANS:32
8 0000000E 7A0200001000	8	MOV.L #DATA:32, ER2
9	9	.FOR.B (R1L=#1, #8, +#1)
10 00000014 F901	S	MOV #1, R1L
11 00000016 5800000A	S	BRA _\$F00002
12 0000001A	S	_\$F00000: .EQU \$
13 0000001A 6828	10	MOV.B @ER2, R0L
14 0000001C 0B02	11	ADDS.L #1, ER2
15 0000001E 5E000000	12	JSR @CHANGE:24
16	13	.ENDF
17 00000022	S	_\$F00001: .EQU \$
18 00000022 8901	S	ADD #1, R1L
19 00000024	S	_\$F00002: .EQU \$
20 00000024 A908	S	CMP #8, R1L
21 00000026 4FF2	S	BLE _\$F00000
22 00000028	S	_\$F00003: .EQU \$
23 00000028 0180	14	SLEEP
24 0000002A 40D4	15	BRA START
25	16	;
26 0000002C	17	CHANGE
27 0000002C 6A2900000000	18	MOV.B @ANS:32, R1L
28	19	.IF.B (R1L<LT>R0L)
29 00000032 1C98	S	CMP R1L, R0L
30 00000034 58F00006	S	BLE _\$I00000
31 00000038 6AA800000000	20	MOV.B R0L, @ANS:32
32	21	.ENDI
33 0000003E	S	_\$I00000: .EQU \$
34 0000003E	S	_\$I00001: .EQU \$
35 0000003E 5470	22	RTS
36	23	;
37 00001000	24	.SECTION BBB, DATA, LOCATE=H'00001000
38 00001000	25	DATA
39 00001000 03020405	26	.DATA.B H'03, H'02, H'04, H'05
40 00001004 01080607	27	.DATA.B H'01, H'08, H'06, H'07
41	28	;
42 00000000	29	.SECTION CCC, DATA, ALIGN=2
43 00000000	30	ANS
44 00000000 00000001	31	.RES.B 1
45	32	;
46 00000000	33	.SECTION DDD, STACK, ALIGN=2
47 00000000 00000500	34	.RES.B H'500
48 00000500	35	STACK
49	36	;
50 00000000	37	.END START

---

(1)      (2)                      (3)      (4) (5)                                      (6)

\*\*\*\*\*TOTAL ERRORS 0

\*\*\*\*\*TOTAL WARNINGS 0

**Figure 8.8 Source Program Listing**

Description

- (1) Line numbers in list
- (2) Value of the location counter
  - Displays absolute address for absolute address section and displays relative address for relative address section.
- (3) Object code
- (4) Source line numbers
  - The line number of source statement in the source program. No line number is displayed for source statements expanded by the assembler.
- (5) Expansion type
  - Source statement of preprocessor function. The following expansion types are available.
    - I: File inclusion
    - C: Satisfied conditional assembly, performed iterated expansion, or satisfied conditional iterated expansion
    - M: Macro expansion
    - S: Structure assembly expansion
- (6) Source statements

8.3.3 Cross Reference Listing

The cross reference listing is shown. Figure 8.9 shows an example of cross reference listing.

*** CROSS REFERENCE LIST				
NAME	SECTION	ATTR	VALUE	SEQUENCE
AAA	AAA	SCT	00000000	3*
ANS	CCC		00000000	7 27 31
				43*
BBB	BBB	SCT	00001000	37*
CCC	CCC	SCT	00000000	42*
CHANGE	AAA		0000002C	15 26*
DATA			00001000	8 38*
DDD	DDD	SCT	00000000	46*
STACK	DDD		00000500	5 48*
START	AAA		00000000	4* 24 50
_\$F00000		AAA	EQU	0000001A 12* 21
_\$F00001	AAA		EQU	00000022 17*
_\$F00002	AAA		EQU	00000024 11 19*
_\$F00003	AAA		EQU	00000028 22*
_\$I00000	AAA		EQU	0000003E 30 33*
_\$I00001	AAA		EQU	0000003E 34*
<hr/>				
( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 )				

Figure 8.9 Cross Reference Listing

Description

- (1) Symbol name

(2) Section name

The name of the section that includes the symbol. Up to eight characters are displayed.

(3) Symbol attribute

No display	Label definition
EQU	Symbol defined with the .EQU assembler directive
ASGN	Symbol defined with the .ASSIGN assembler directive
IMPT	Import symbol
EXPT	Export symbol
SCT	Section name
REG	Symbol defined with the .REG assembler directive
MDEF	Symbol defined two or more times
UDEF	Undefined symbol

(4) Symbol value

The hexadecimal value of a symbol in eight digits

(5) List line numbers of symbol definition or reference

The list line numbers of the source statements where the symbol is defined or referenced. The line number marked with an asterisk (\*) is the line where the symbol is defined.

### 8.3.4 Section Information Listing

The section information listing is shown. Figure 8.10 shows an example of section information listing.

*** SECTION DATA LIST			
SECTION	ATTRIBUTE	SIZE	START
AAA	REL-CODE	0000040	
BBB	ABS-DATA	0000008	001000
CCC	REL-DATA	0000001	
DDD	REL-STACK	0000500	
<hr/>			
(1)	(2)	(3)	(4)

**Figure 8.10 Section Information Listing Output Example**

#### Description

(1) Section name

(2) Section type and attribute

The section type and attribute are shown below:

— Section type

ABS Absolute address section

REL Relative address section

— Section attribute

CODE Code section

DATA Data section

STACK Stack section

DUMMY Dummy section

(3) Section size

The section size is displayed in hexadecimal.

(4) Section start address

The start address of absolute address sections. This will not be displayed in the relative address sections.

## 8.4 Linkage Listings

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

## 8.4.1 Structure of Linkage Listing

Table 8.5 shows the structure and contents of the linkage listing.

**Table 8.5 Structure and Contents of Linkage Listing**

Information Creating List	Contents	Suboption	Default When show Option Omitted*1
Option information	Displays option strings specified by a command line or subcommand	—	Output
Error information	Displays error messages	—	Output
Linkage map information	Displays a section name, start and end addresses, size, and type	—	Output
Symbol information	Displays static definition symbol name, address, size, and type in order based on the address.	show= symbol	Not output
	When the <b>show=reference</b> option is specified, displays a symbol reference count and optimization information in addition to the above information.	show= reference	Not output
Symbol deletion optimization information	Displays symbols deleted by optimization	show= symbol	Not output
Variable access optimization symbol information	Displays symbol reference counts in 8-bit/16-bit absolute addressing mode.	show= reference	Not output
Function access optimization symbol information	Displays symbol reference counts.	show= reference	Not output
Cross-reference information	Displays symbol reference information	show = xreference	Not output

Note: 1. The **show** option is valid only when the **list** option is specified.



## 8.4.2 Option Information

Option information displays option strings specified by a command line or a subcommand file. The option information is output as shown in figure 8.11 when **optlink -sub=test.sub -list -show** is specified.

```
(Contents of test.sub)
```

```
INPUT test.obj
```

```
*** Options ***
```

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

**Figure 8.11 Option Information Output Example (Linkage Listing)**

Description

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

## 8.4.3 Error Information

Error information outputs an error message as shown in figure 8.12.

```
*** Error information ***
```

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

**Figure 8.12 Error Information Output Example (Linkage Listing)**

Description

- (1) Error message

#### 8.4.4 Linkage Map Information

Linkage map information outputs the start and end addresses, size, and type of each section in order of addresses in the format shown in figure 8.13.

*** Mapping List ***				
<u>SECTION</u> (1)	<u>START</u> (2)	<u>END</u> (3)	<u>SIZE</u> (4)	<u>ALIGN</u> (5)
P	00000000	000004d6	4d6	2
C	000004d6	00000533	5d	2
D	00000534	0000053c	8	2
B	0000053c	00004112	3bd6	2

**Figure 8.13 Linkage Map Information Output Example  
(Linkage Listing)**

Description

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

#### 8.4.5 Symbol Information

When the **show=symbol** option is specified, symbol information lists addresses of externally defined symbols or static internally defined symbols, sizes, and types in order of address. When the **show=reference** option is specified, symbol information lists symbol reference counts and optimization information in addition to the information listed when the **show=symbol** option is specified. Figure 8.14 shows an example of symbol information.

*** Symbol List ***						
SECTION=(1)						
FILE=(2)						
	<u>START</u>	<u>END</u>	<u>SIZE</u>			
	(3)	(4)	(5)			
<u>SYMBOL</u>	<u>ADDR</u>	<u>SIZE</u>	<u>INFO</u>	<u>COUNTS</u>	<u>OPT</u>	
(6)	(7)	(8)	(9)	(10)	(11)	
SECTION=P						
FILE=test.obj						
	00000000	00000428	428			
_main						
	00000000	2	func ,g	0		
_malloc						
	00000000	32	func ,l	0		
FILE=mvn3						
	00000428	00000490	68			
\$MVN#3						
	00000428	0	none ,g	0		

**Figure 8.14 Symbol Information Output Example (Linkage Listing)**

#### Description

- (1) Section name
- (2) File name
- (3) Start address of a section included in the file in (2) above
- (4) End address of a section included in the file in (2) above
- (5) Section size of a section included in the file in (2) above
- (6) Symbol name
- (7) Symbol address
- (8) Symbol size
- (9) Symbol type as shown below:
 

Data type:	func	Function name
	data	Variable name
	entry	Entry function name
	none	Undefined (label, assembler symbol)
Declaration type:	g	External definition
	l	Internal definition
- (10) Symbol reference count only when the **show=reference** option is specified. \* is displayed when the **show=reference** option is not specified.
- (11) Optimization information as shown below:
 

ch	Symbol modified by optimization
cr	Symbol created by optimization
mv	Symbol moved by optimization

8.4.6 Symbol Deletion Optimization Information

Symbol deletion optimization information lists the size and type of symbols deleted by symbol deletion optimization (**optimize=symbol\_delete**) as shown in figure 8.15.

*** Delete Symbols ***		
<u>SYMBOL</u>	<u>SIZE</u>	<u>INFO</u>
(1)	(2)	(3)
_Version	4	data ,g

Figure 8.15 Symbol Deletion Information Output Example (Linkage Listing)

Description

- (1) Deleted symbol name
- (2) Deleted symbol size
- (3) Deleted symbol type as shown below

Data type:	func	Function name
	data	Variable name
Declaration type:	g	External definition
	l	Internal definition

8.4.7 Variable Access Optimization Symbol Information

When the **show=reference** option is specified, variable access optimization symbol information lists the size, reference count, and optimization information of the symbol to be optimized on variable access optimization (**optimize=variable\_access**).

Information of symbols that can be accessed in 8-bit or 16-bit absolute addressing mode is listed in the area "Variable Accessible with Abs8". Information of symbols that can be accessed in 16-bit absolute addressing mode is listed in the area "Variable Accessible with Abs16".

Figure 8.16 shows an example of variable access optimization symbol information.

```

*** Variable Accessible with Abs8 ***

SYMBOL          SIZE      COUNTS  OPTIMIZE
(1)             (2)       (3)      (4)
_Char1Glob
                1         2      done

*** Variable Accessible with Abs16 ***

SYMBOL          SIZE      COUNTS  OPTIMIZE
(1)             (2)       (3)      (4)
_IntGlob
                2         2

```

**Figure 8.16 Output Example of Variable Access Optimization Symbol Information (Linkage Listing)**

#### Description

- (1) Symbol name
- (2) Symbol size
- (3) Symbol reference count
- (4) Optimization information.

If optimization has been performed, "done" is displayed.

### 8.4.8 Function Access Optimization Symbol Information

When the **show=reference** option is specified, function access optimization symbol information lists the reference count and optimization information of the symbol to be optimized on function access optimization (**optimize=function\_call**).

Figure 8.17 shows an example of function access optimization symbol information.

*** Function Call ***		
<u>SYMBOL</u>	<u>COUNTS</u>	<u>OPTIMIZE</u>
(1)	(2)	(3)
_malloc	5	done
_Proc0	4	

**Figure 8.17 Output Example of Function Access Optimization Symbol Information (Linkage Listing)**

Description

- (1) Symbol name
- (2) Symbol reference count
- (3) Optimization information.

If optimization is performed, "done" is displayed.

## 8.4.9 Cross-Reference Information

The symbol reference information (cross-reference information) can be output. A cross-reference information output example is shown in figure 8.18.

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

No      Unit Name      Global.Symbol      Location      External Information
(1)      (2)          (3)              (4)          (5)
0001 a
SECTION=P
      _func
      00000100
      _func1
      00000116
      _main
      0000012c
      _g
      00000136
SECTION=B
      _a
      00000190 0001(00000140:P)
      0002(00000178:P)
      0003(0000018c:P)
0002 b
SECTION=P
      _func01
      00000154 0001(00000148:P)
      _func02
      00000166 0001(00000150:P)
0003 c
SECTION=P
      _func03
      00000184
```

**Figure 8.18 Cross-Reference Information Output Example (Linkage Listing)**

### Description:

- (1) Unit number, which is an identification number in object units
- (2) Object name, which specifies the input order at linkage
- (3) Symbol name output in ascending order for every section
- (4) Symbol allocation address, which is a relative value from the beginning of the section when **form=rel** is specified
- (5) Address from which an external symbol is referenced  
Output format: <Unit number> (<address or offset in section>:<section name>)

## 8.5 Library Listings

This section covers the contents and format of the library listing output by the optimization linkage editor.

### 8.5.1 Structure of Library Listing

Table 8.6 shows the structure and contents of the library listing.

**Table 8.6 Structure and Contents of Library Listing**

List Structure	Contents	Suboption	Default When show Option Omitted* <sup>1</sup>
Option information	Displays option strings specified by a command line or subcommand	—	Output
Error information	Displays error messages	—	Output
Library information	Displays library information	—	Output
Information of module, section, and symbol within library	Displays module within the library		Output
	When the <b>show=symbol</b> option is specified, displays a list of symbol names in a module.	show=symbol	Not output
	When the <b>show=section</b> option is specified, displays a list of section names and symbol names in a module in addition to the above information.	show=section	Not output

Note: 1. The **show** option is valid only when the **list** option is specified.



### 8.5.2 Option Information

Option information displays option strings specified by a command line or a subcommand file. Figure 8.19 shows an example of option information when **optlnk -sub=test.sub -list -show** is specified.

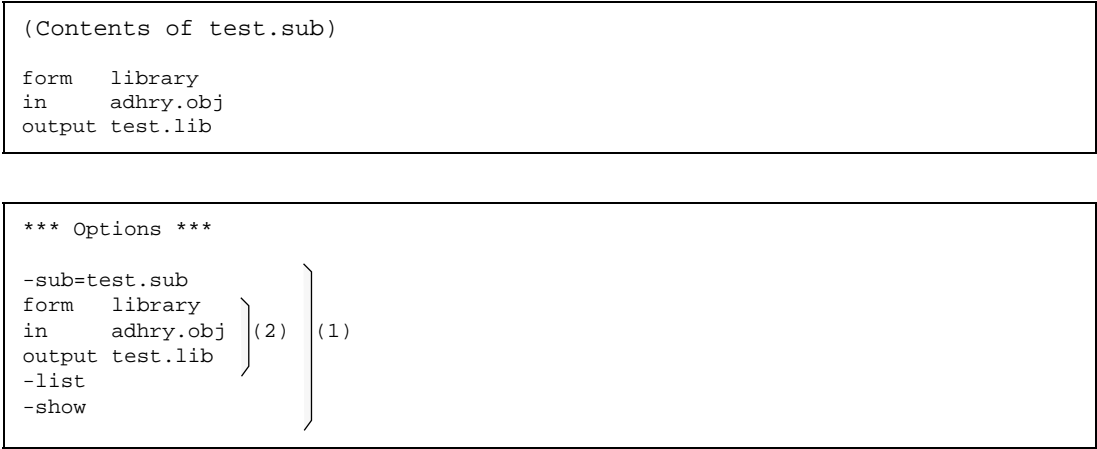


Figure 8.19 Option Information Output Example (Library Listing)

Description

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

### 8.5.3 Error Information

Error information outputs an error message as shown in figure 8.20.

```
*** Error information ***  
  
** L1200 (W) Backed up file "main.lib" into "main.lbk" (1)
```

**Figure 8.20 Error Information Output Example (Library Listing)**

Description

(1) Error message

### 8.5.4 Library Information

Library information outputs the library type in the format shown in figure 8.21.

```
*** Library Information ***  
  
LIBRARY NAME=test.lib (1)  
CPU=H8S (2)  
ENDIAN=Big (3)  
ATTRIBUTE=system (4)  
NUMBER OF MODULE=1 (5)
```

**Figure 8.21 Library Information Output Example (Library Listing)**

Description

(1) Library name

(2) CPU name

(3) Endian type

(4) Library file attribute as either system library or user library

(5) Number of modules within the library

### 8.5.5 Module, Section, and Symbol Information within Library

This information lists modules within the library.

When the **show=symbol** option is specified, symbol names in a module within the library are listed. When the **show=section** option is specified, section names and symbol names in a module within the library are additionally listed.

Figure 8.22 shows an output example of module, section, and symbol information within a library.

```
*** Library List ***

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

**Figure 8.22 Output Example of Module, Section, and Symbol Information within Library (Library Listing)**

Description

- (1) Module name
- (2) Module definition date  
If the module is updated, the latest module update date is displayed.
- (3) Section name within a module
- (4) Symbol within a section



# Section 9 Programming

## 9.1 Program Structure

### 9.1.1 Sections

Each of the regions for execution instructions and data of the object programs output by the C/C++ compiler or 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
stack	Stack area
- 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 one initial value, the remaining area without initial values is initialized to zero.
- Write operations

Specifies whether write operations are or are not possible during program execution.
- Boundary alignment

Corrections to addresses assigned to sections. The optimizing linkage editor corrects addresses such that they are multiples of the boundary alignment.

### 9.1.2 C/C++ Program Sections

The correspondence between standard library memory areas and sections for C/C++ programs is described in table 9.1.

**Table 9.1 Summary of Memory Area Types and Their Properties**

Name	Section		Format Type	Initial Values		Align-ment	Description
	Name	Attribute		Write Operations			
Program area	P* <sup>1</sup>	code	Relative	Yes No		2 bytes	Stores machine code
Constant area	C* <sup>1</sup>	data	Relative	Yes No		2 bytes	Stores const-type data
Initialized data area	D* <sup>1</sup>	data	Relative	Yes Yes		2 bytes	Stores data with initial values
Uninitialized data area	B* <sup>1</sup>	data	Relative	No Yes		2 bytes	Stores data without initial values
Constant area (8-bit address space)	\$ABS8C* <sup>1</sup>	data	Relative	Yes No		1 byte	Stores const-type 8-bit data specified by the abs8 option, or by __abs8, #pragma abs8
Initialized data area (8-bit address space)	\$ABS8D* <sup>1</sup>	data	Relative	Yes Yes		1 byte	Stores 8-bit data with initial values specified by the abs8 option, or by __abs8, #pragma abs8
Uninitialized data area (8-bit address space)	\$ABS8B* <sup>1</sup>	data	Relative	No Yes		1 byte	Stores 8-bit data without initial values specified by the abs8 option, or by __abs8, #pragma abs8
Constant area (16-bit address space)	\$ABS16C* <sup>1</sup>	data	Relative	Yes No		2 bytes	Stores const-type data specified by the abs16 option, or by __abs16, #pragma abs16
Initialized data area (16-bit address space)	\$ABS16D* <sup>1</sup>	data	Relative	Yes Yes		2 bytes	Stores data with initial values specified by the abs16 option, or by __abs16, #pragma abs16
Uninitialized data area (16-bit address space)	\$ABS16B* <sup>1</sup>	data	Relative	No Yes		2 bytes	Stores data without initial values specified by the abs16 option, or by __abs16, #pragma abs16

**Table 9.1 Summary of Memory Area Types and Their Properties (cont)**

Name	Section		Format Type	Initial Values		Align-ment	Description
	Name	Attribute		Write Operations			
Function address area (memory indirect space)	\$INDIRECT* <sup>1</sup>	data	Relative	Yes No		2 bytes	Stores function addresses specified by the indirect=normal option, or by <code>_indirect</code> , <code>#pragma indirect</code>
Function address area (extended memory indirect space)	\$EXINDIRECT* <sup>1</sup>	data	Relative	Yes No		2 bytes	Stores function addresses specified by the indirect=extended option, or by <code>_indirect_ex</code>
Function address area (memory indirect space)	\$VECTxx* <sup>1</sup> xx: vector number	data	Absolute	Yes No		2 bytes	Stores function addresses specified with vect=xx of <code>_indirect</code> , <code>#pragma indirect</code> , <code>_indirect_ex</code> , <code>_interrupt</code> , <code>#pragma interrupt</code> , <code>_entry</code> , or <code>#pragma entry</code>
1-byte data area	yy\$1* <sup>2</sup> yy:C* <sup>1</sup> ,D* <sup>1</sup> ,B* <sup>1</sup> , \$ABS16C* <sup>1</sup> , \$ABS16D* <sup>1</sup> , \$ABS16B* <sup>1</sup>	data	Relative	—		1 byte	Handles 1-byte data when the align=4 option is specified, and is created in each section
4-byte data area	yy \$4* <sup>2</sup> yy:C* <sup>1</sup> ,D* <sup>1</sup> ,B* <sup>1</sup> , \$ABS16C* <sup>1</sup> , \$ABS16D* <sup>1</sup> , \$ABS16B* <sup>1</sup>	data	Relative	—		4 bytes	Handles 4-byte data when the align=4 option is specified, and is created in each section

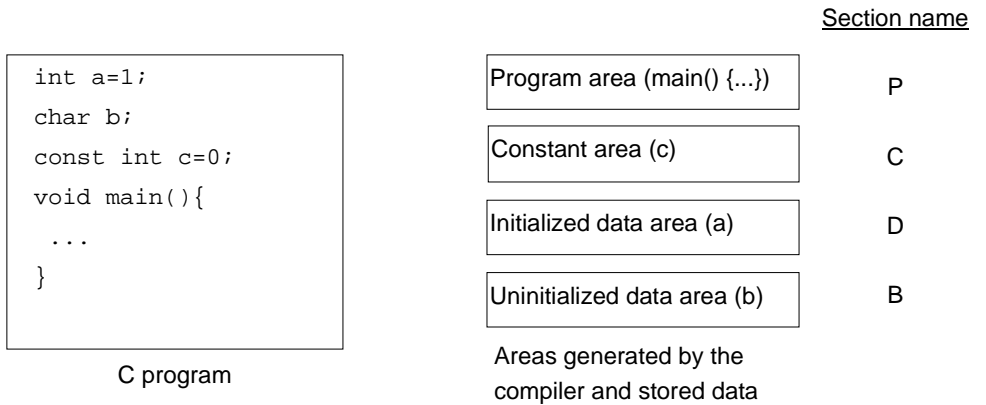
**Table 9.1 Summary of Memory Area Types and Their Properties (cont)**

Name	Section		Format Type	Initial Values		Alignment	Description
	Name	Attribute		Write Operations			
Address area for initialized data section	C\$DSEC* <sup>3</sup>	data	Relative	Yes No		2 bytes	Stores ROM addresses, final addresses in ROM, and RAM addresses for initialized data area sections
Address area for uninitialized data section	C\$BSEC* <sup>3</sup>	data	Relative	Yes No		2 bytes	Stores addresses and final addresses for uninitialized data area sections
C++ initial processing/postprocessing data area	C\$INIT* <sup>3</sup>	data	Relative	Yes No		2 bytes	Stores addresses of constructors and destructors called for global class objects
C++ virtual function table area	C\$VTBL* <sup>3</sup>	data	Relative	Yes No		2 bytes	Stores data for virtual function calls when there is a virtual function in a class declaration
Stack area	S	stack	Relative	No Yes		2 bytes	Area necessary for program execution (see section 9.2.1 (2), Dynamic Area Allocation)
Heap area	—	—	Relative	No Yes		—	Area used by library functions malloc, realloc, calloc, new (see section 9.2.1 (2), Dynamic Area Allocation)
Absolute address variable area	\$ADDRESS \$yy<address> yy:C,D,B	data	Absolute	Yes/No Yes/No* <sup>4</sup>		—	Stores variables specified by #pragma address

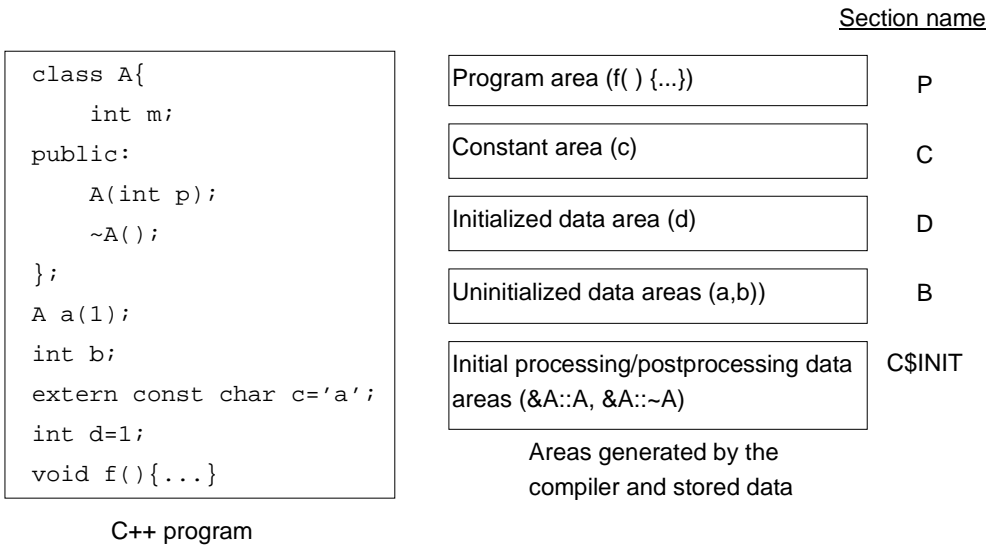
- Notes: 1. Section names can be switched in the compiler option section, extension #pragma section, #pragma abs8 section, #pragma abs16 section, or #pragma indirect section.
2. The data section name before data subdivision is to be displayed in place of yy.  
e.g. C -> C\$1,C\$.
3. When the compiler option section=C=zz is specified, the prefix "C" becomes "zz".
4. The initial value and write operation depend on the attributes of sections C, D, and B.



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



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



### 9.1.3 Assembly Program Sections

In assembly programs, .SECTION directives are used to begin sections and declare attributes and formats. The format for declaration of a .SECTION directive is given below. For details, refer to section 11.3, Assembler Directives.

.SECTION <section name>[,<section attribute>[,<format type>]]  
<format type>:        In the case of a relative address section, align = <alignment boundary>  
                      In the case of an absolute address section, locate = <address value>

Example: An example of an assembly program section declaration appears below.

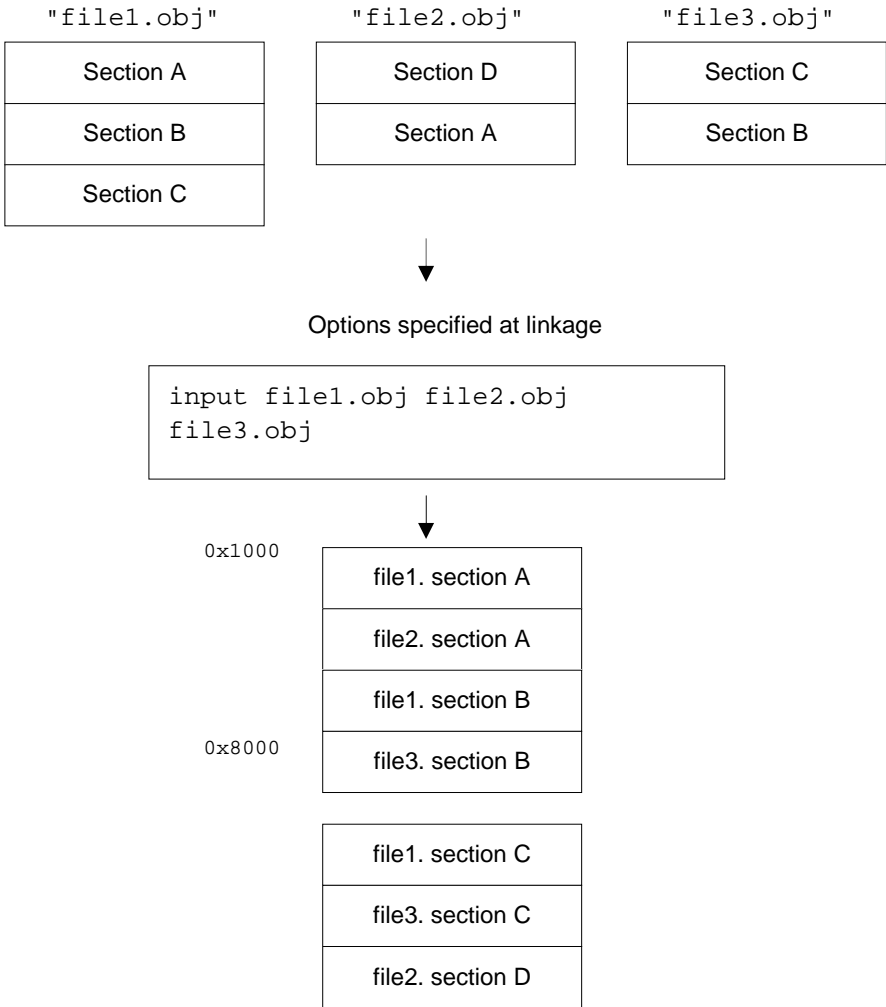
```
.CPU 2600A
.OUTPUT DBG
SIZE : .EQU 8
;
.SECTION A, CODE, ALIGN=2      ..... (1)
START:
    MOV.L #CONST:32, ER0
    MOV.L #DATA:32, ER1
    MOV.L #SIZE:32, ER2
LOOP:
    CMP.L #0:32, ER2
    BEQ EXIT
    MOV.B @ER0, R3L
    MOV.B R3L, @ER1
    ADD.L #1:32, ER0
    ADD.L #1:32, ER1
    SUB.L #1:32, ER2
    BRA LOOP
EXIT:
    SLEEP
    BRA START
;
.SECTION B, DATA, LOCATE=H'00001000 ..... (2)
CONST
    .DATA.B H'01, H'02, H'03, H'04
    .DATA.B H'05, H'06, H'07, H'08
;
.SECTION C, STACK, ALIGN=2     ..... (3)
DATA
    .RES.B SIZE
;
.END START
```

- (1) Declares a code section with section name A, alignment boundary 2, and relative address format.
- (2) Declares a data section with section name B, allocated address H'1000, and absolute address format.
- (3) Declares a stack section with section name C, alignment boundary 2, and relative address format.

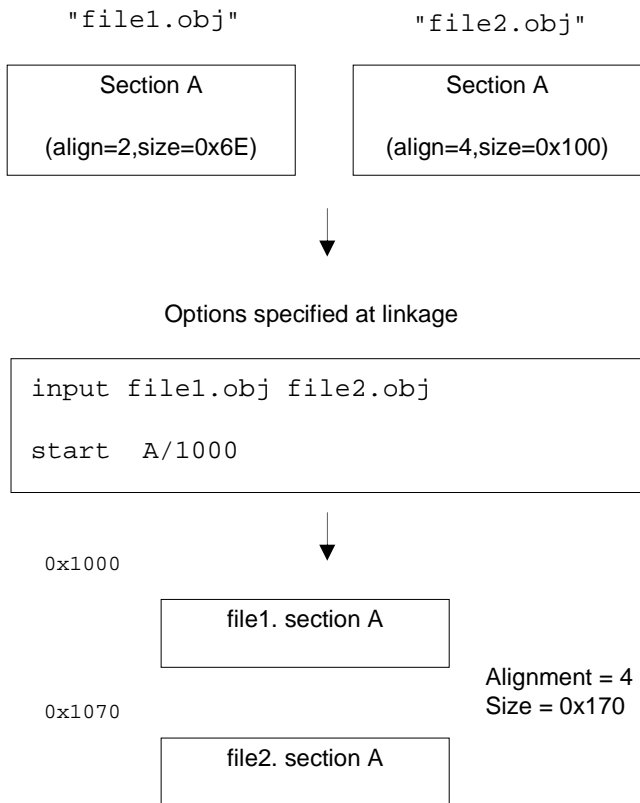
### 9.1.4     Linking Sections

The optimizing linkage editor links the same sections within input object programs, and allocates addresses specified using the **start** option.

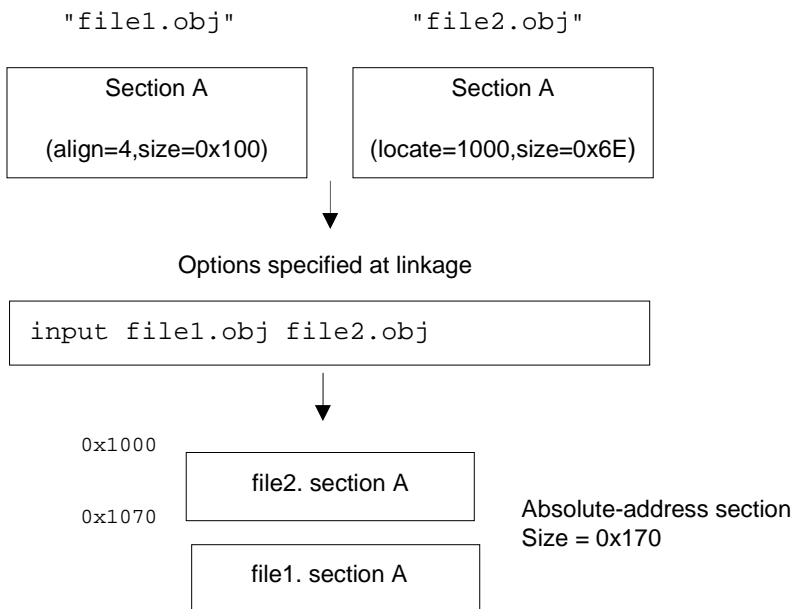
- (1) The same section names in different files are allocated continuously in the order of file input.



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

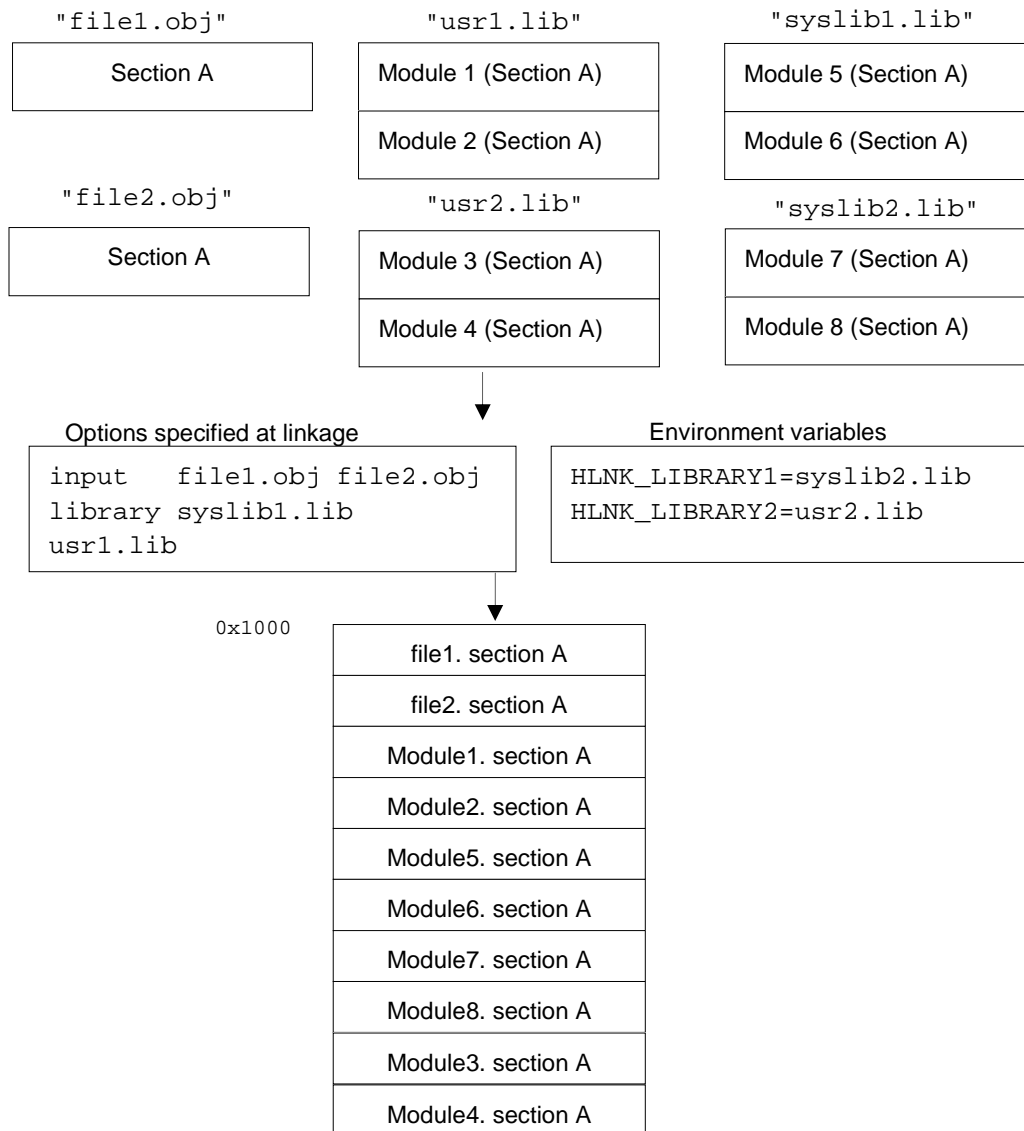


- (3) When sections with the same name include both absolute-address and relative-address formats, relative-address objects are linked following absolute-address objects. Even when relocatable file (form=relocate) output is specified, the section in question becomes an absolute-address section.



(4) Rules for the order of linking objects within the same section name are as follows.

- Order specified by the **input** option or in the order of 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



## 9.2 Creation of Initial Setting Programs

Here methods for embedding programs into systems employing the H8SX, AE5, H8S/2600, H8S/2000, H8/300H and H8/300 are explained.

To embed a program in a system, the following preparations are necessary.

- **Memory allocation**  
Each section, the stack area, and the heap area must be allocated to system ROM and RAM.
- **Settings for the program execution environment**  
Processing to set the program execution environment includes register initialization, memory initialization, and program startup.

In addition, when using I/O and other C/C++ library functions, the library must be initialized during preparation of the execution environment. In particular, when using I/O (stdio.h, ios, streambuf, istream, ostream) and memory allocation (stdlib.h, new), low-level I/O routines and memory allocation routines must be created.

When using C library functions for program termination (the exit, atexit, abort functions), these functions must be created separately according to the user system.

In section 9.2.1, the method used to determine addresses for program memory is explained, and actual examples are used to describe the method for specifying options in the optimizing linkage editor for determining addresses.

In section 9.2.2, execution environment settings are explained, and an actual example of a program to set the execution environment is described.

Library function initialization processing, creation of low-level routines, and examples of creation of functions for termination processing are also explained.

### 9.2.1 Memory Allocation

In order to embed an object program into a system, the size of the memory areas to be used by the program must be determined, and these memory areas must be allocated to appropriate memory addresses.

Memory areas used by a program include areas which are statically allocated, such as for execution instructions corresponding to functions in the program and data declared using external data definitions, and areas which are dynamically allocated, such as the stack area. Below, methods for allocation of each type of area are explained.

## (1) Static memory area allocation

### (a) Contents of static memory area

Sections other than the stack area and heap area are allocated statically.

Each of the sections in a C/C++ program (program area, constant area, initialized data area, uninitialized data area, function address area, initialized data section address area, uninitialized data section address area, C++ initial processing/postprocessing data area, and C++ virtual function table area) is allocated statically.

### (b) Calculation of size

The size of static memory is the sum of the sizes of the object programs generated by the compiler and assembler and the sizes of the library functions used by the C/C++ program.

After linking an object program, the sizes of each section, including libraries, are output to the linkage map information of the linkage list, and so the size of static memory can be determined. Figure 9.1 shows an example of linkage map information in the linkage list.

\*\*\* Mapping List \*\*\*

<u>SECTION</u> (1)	<u>START</u> (2)	<u>END</u> (3)	<u>SIZE</u> (4)	<u>ALIGN</u> (5)
P	00000000	000004d6	4d6	2
C	000004d6	00000533	5d	2
D	00000534	0000053c	8	2
B	0000053c	00004112	3bd6	2

**Figure 9.1 Example of Linkage Map Information in Linkage List**

Section sizes of compiling and assembly units are output to the statistics information of the compile list and section information of the assembly list. An example of compile list statistics information is shown in figure 9.2, and an example of assembly list section information appears in figure 9.3.

\*\*\*\*\* SECTION SIZE INFORMATION \*\*\*\*\*

PROGRAM	SECTION (P):	0x00000080	Byte(s)
CONSTANT	SECTION (C):	0x00000004	Byte(s)
DATA	SECTION (D):	0x00000004	Byte(s)
BSS	SECTION (B):	0x00000004	Byte(s)

TOTAL	PROGRAM	SECTION:	0x00000080	Byte(s)
TOTAL	CONSTANT	SECTION:	0x00000004	Byte(s)
TOTAL	DATA	SECTION:	0x00000004	Byte(s)
TOTAL	BSS	SECTION:	0x00000004	Byte(s)

TOTAL	PROGRAM	SIZE:	0x0000008C	Byte(s)
-------	---------	-------	------------	---------

**Figure 9.2 Example of Compile List Statistics Information**



*** SECTION DATA LIST			
SECTION	ATTRIBUTE	SIZE	START
P	REL-CODE	000000604	
D	REL-DATA	000000008	
C	REL-DATA	00000005D	
B	REL-DATA	000003BD6	

**Figure 9.3 Example of Assembly List Section Information**

When not using a standard library, the total of file-unit section sizes is the size of static memory.

When using a standard library, memory area sizes used by library functions must be added to the memory size for each section. Among the standard libraries provided by the compiler are, in addition to C library functions stipulated by the C language specifications and C++ class libraries for embedded use, routines to perform arithmetic calculations (runtime routines) used for program execution. Hence even if use of library functions is not specified in the source program, a standard library may be needed.

The runtime routines used by a program can be determined from the symbol allocation information in the compile list output by the compiler. A specific example is presented below.

C program

```
long a,b;
main()
{
    a *= b;
}
```

C compiler output symbol allocation information

\*\*\*\*\* STACK FRAME INFORMATION \*\*\*\*\*

FILE NAME: main.c

Function (File main.c , Line 2):main

```
Parameter Area Size      : 0x00000000 Byte(s)
Linkage Area Size        : 0x00000000 Byte(s)
Local Variable Size      : 0x00000000 Byte(s)
Temporary Size           : 0x00000000 Byte(s)
Register Save Area Size  : 0x00000000 Byte(s)
Total Frame Size         : 0x00000000 Byte(s)
```

## (c) ROM, RAM allocation

When writing a program to ROM, whether sections are allocated to RAM or to ROM is determined by whether there are initial values and whether write operations are enabled.

When writing the sections of a C/C++ program to ROM, sections are allocated to ROM or to RAM as follows.

• Program area (section P)	ROM
• Constant areas (sections C, \$ABS8C, \$ABS16C)	ROM
• Uninitialized data areas (sections B, \$ABS8B, \$ABS16B)	RAM
• Initialized data areas (sections D, \$ABS8D, \$ABS16D)	ROM, RAM
(see (d) below)	
• Function address area (section \$INDIRECT, \$EXINDIRECT)	ROM
• Initialized data section address area (section C\$DSEC)	ROM
• Uninitialized data section address area (section C\$BSEC)	ROM
• Initial processing data area* <sup>1</sup> (section C\$INIT)	ROM
• Virtual function table area* <sup>2</sup> (section C\$VTBL)	ROM

- Notes: 1. Generated by the compiler when a C++ program has a global class object.  
 2. Generated by the compiler when a C++ program contains virtual function declarations.

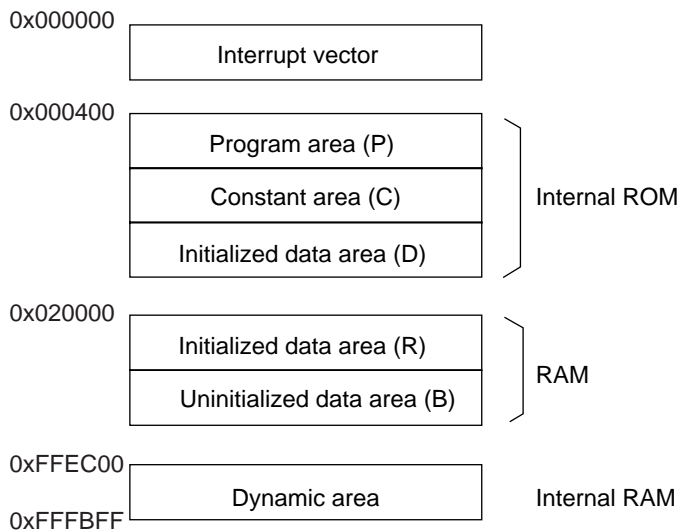
## (d) Allocation of initialized data areas

Sections which have initial values and can be altered on program execution, such as initialized data areas, are placed in ROM at link time and copied to RAM at the start of program execution. Hence the rom option of the optimizing linkage editor must be used to reserve the duplicate memory area both in ROM and in RAM. For an example of this, refer to "(e) Example of memory allocation and address specification at link time" below. Initial settings for sections to be copied from ROM to RAM are explained in section 9.2.2 (2), Initial settings (PowerON\_Reset).

## (e) Example of memory allocation and address specification at link time

When creating an absolute load module, addresses are specified per allocated area for each section using an optimizing linkage editor option or a subcommand. Below, examples of static memory allocation and of address specification at link time are explained.

Figure 9.4 shows an example of allocation of a static memory area in H8S/2600 advanced mode.



P, C, D, B: Default section names generated by the compiler.

R: Section name specified by ROM support function of the linkage editor.

**Figure 9.4 Example of Static Memory Allocation**

When allocating memory as shown in figure 9.4, the following subcommands are specified at link time.

```
ROMAD=R ... [1]
STARTAP,C,D/400,R,B/20000 ... [2]
```

**Explanation [1]** Space for section R of size equal to that of section D is secured in the output load module. When symbols allocated to section D are referenced, relocation is performed so that their addresses are in section R. Section D and section R are initialized data sections on ROM and to RAM respectively.

**Explanation [2]** Sections P, C and D are allocated to contiguous areas of memory in internal ROM starting from address 0x400. Sections R and B are allocated to contiguous memory areas starting from address 0x20000 in RAM.

## (2) Dynamic memory area allocation

### (a) Contents of dynamic memory

The following two types of dynamic memory areas are used in C/C++ programs:

- Stack area
- Heap area (for memory allocation of library functions and other uses)

## (b) Calculation of stack area size

The maximum stack area size used by C/C++ programs and standard libraries can be calculated by specifying the stack option of the optimizing linkage editor to output a stack information file, and using the stack usage analysis tool. For details of use of the stack usage analysis tool, refer to section 6, Operating Stack Analysis Tool.

The stack analysis tool can calculate the stack usage, if label is specified by .STACK directive. But it cannot calculate the stack area used by an assembly program, which was assembled by the assembler unable to output to a stack information file. Instead, the stack usage of an assembly program should be computed by the method outlined below for calculating the stack usage of a C/C++ program, and the result should be added to the stack usage calculated by the stack usage analysis tool.

**Method for Calculating Stack Usage by C/C++ Program:** Stack area is allocated for use by a C/C++ program each time a function is called, and is released when the function returns. In order to calculate the size of the stack area used, first the amount of stack space used by each function is computed, and then the calling relations of functions are used to calculate the actual stack space use.

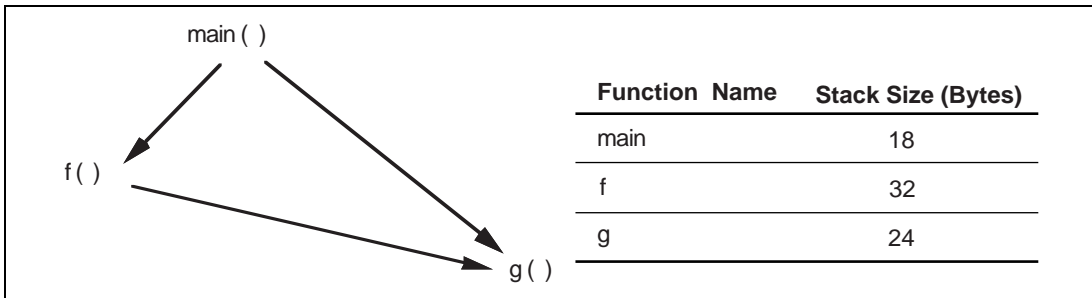
The stack area used by each function can be found from the symbol allocation information (total frame size) of the compile list.

```
***** STACK FRAME INFORMATION *****  
FILE NAME: test.c  
Function (File test.c      , Line    2):main  
    Optimize Option Specified : No Allocation Information Available  
Paramater Area Size          : 0x00000008 Byte(s)  
Linkage Area Size            : 0x00000004 Byte(s)  
Local Variable Size          : 0x00000002 Byte(s)  
Temporary Size               : 0x00000000 Byte(s)  
Register Save Area Size      : 0x00000004 Byte(s)  
Total Frame Size             : 0x00000012 Byte(s)
```

The stack area used by each function can be found from the symbol allocation information (total frame size) of the compile list.

The stack area used by the function is the total frame size of 0x12, that is, 18 bytes.

An example of function calling relationships and stack use by each function appears in figure 9.5. Here, the size of the stack used when function g is called via function f is calculated in table 9.2.



**Figure 9.5 Example of Function Calling Relationships and Stack Area Used**

**Table 9.2 Example of Calculation of Stack Area Used**

Calling Path	Stack Area Used	Remarks
main (18) → f (32) → g(24)	74	Stack space used (maximum)
main (18) → g(24)	42	

In this way, the stack area used is calculated for the function at the deepest calling level, and stack area for this maximum value (in this case, 74 bytes) is allocated.

#### Note on stack consumption calculation

The fundamental to calculate the amount of stack consumption differs between Ver. 4.0 or earlier or Ver. 6.0 except for H8SX and H8SX of Ver. 6.0. In this note, Ver. 4.0 or earlier and Ver. 6.0 except for H8SX is called the group A, and H8SX of Ver. 6.0 and H8S or H8SX of Ver. 6.01 is called the group B. Take care if a function compiled by the group A calls a function compiled by the group B, and vice versa.

The behavior of the SP, the stack pointer, differs between the group A and B. In the group A, a parameter passed via the stack is stored after decrementing the SP using the push instruction or the pre-decrement addressing mode (@-SP) as shown at [1] of the following example. After the return from the function call, the stack area for the parameter is released through incrementing the SP by the parameter size as shown at [2] of the following example. In group A, the size of the parameter area in the stack differs depending on a function, and that size is counted into the Parameter Area Size of the callee's stack frame size as shown at [3] of the following example.

On the other hand, in the group B, the compiler calculates the maximum amount of the stack area used in the function beforehand, and that amount of stack area is reserved at the function prolog as shown at [4] of the following example. The SP is unchanged until the function epilog, and the SP is restored to the original value before the function itself is called, as shown at [6] of the following example. In this case, a parameter is stored at an address with 0 or positive offset from the SP without changing the SP, as shown at [5] of the following example. In group B, the size including the maximum amount of parameter usage of all the function calls is counted into the Temporary Size of the caller's stack frame size as shown at [7] of the following example

As shown at CASE 1 and CASE 4 below, if the groups of the caller and the callee are the same, the total size of stack consumption for the function g to call the function f is exactly 12 bytes through summing up the Total Frame Size of g and f. As in CASE 2 below, if a function of the group A calls that of the group B, the total size of stack consumption for the function g to call the function f is mistakenly 8 bytes through summing up the Total Frame Size of g and f. This underestimate of the stack consumption came from the fact that the size for the parameter area in the stack is not summed up. As in CASE 3 below, if a function of the group B calls that of the group A, the total size of stack consumption for the function g to call the function f is mistakenly 16 bytes through summing up the Total Frame Size of g and f. This overestimate of the stack consumption came from the fact that the size for the parameter area in the stack is summed up twice.

In order to avoid such underestimate or overestimate, do not mix the group A and B, or correct the estimate of stack consumption finding out the point where a function of the group A calls that of the group B or the point where a function of the group B calls that of the group A.

The amount of stack consumption:

CASE 1: the function g of the group A calls the function f of the group B:  $8 + 4 = 12$

CASE 2: the function g of the group A calls the function f of the group A:  $4 + 4 = 8$

CASE 3: the function g of the group B calls the function f of the group B:  $8 + 8 = 16$

CASE 4: the function g of the group B calls the function f of the group A:  $8 + 4 = 12$

Example:

Source program	The group A	The group B
int f(struct S);	<code>_f:</code>	<code>_f:</code>
void g(void);	<code>SUB.W R0,R0</code>	<code>SUB.W R0,R0</code>
struct S{long p;} st;	<code>RTS</code>	<code>RTS</code>
int x;	<code>_g:</code>	<code>_g:</code>
int f(struct S s){		<code>ADD.W #-4:16,R7 ;[4]</code>
return 0;	<code>MOV.L @_st:32,ER0</code>	<code>MOV.L @_st:32,ER0</code>
}	<code>PUSH.L ER0 ;[1]</code>	<code>MOV.L ER0,@SP ;[5]</code>
void g(void)	<code>BSR _f:8</code>	<code>BSR _f:8</code>
{	<code>ADDS.L #4,SP ;[2]</code>	<code>MOV.W R0,@_x:32</code>
x=f(st);	<code>MOV.W R0,@_x:32</code>	<code>ADDS.L #4,SP ;[6]</code>
}	<code>RTS</code>	<code>RTS</code>
Function f:		
Parameter Area Size	: 0x00000004 Byte(s)[3]	0x00000000 Byte(s)
Linkage Area Size	: 0x00000004 Byte(s)	0x00000004 Byte(s)
Local Variable Size	: 0x00000000 Byte(s)	0x00000000 Byte(s)
Temporary Size	: 0x00000000 Byte(s)	0x00000000 Byte(s)
Register Save Area Size	: 0x00000000 Byte(s)	0x00000000 Byte(s)
Total Frame Size	: 0x00000008 Byte(s)	0x00000004 Byte(s)
Function g:		
Parameter Area Size	: 0x00000000 Byte(s)	0x00000000 Byte(s)
Linkage Area Size	: 0x00000004 Byte(s)	0x00000004 Byte(s)
Local Variable Size	: 0x00000000 Byte(s)	0x00000000 Byte(s)
Temporary Size	: 0x00000000 Byte(s)	0x00000004 Byte(s)[7]
Register Save Area Size	: 0x00000000 Byte(s)	0x00000000 Byte(s)
Total Frame Size	: 0x00000004 Byte(s)	0x00000008 Byte(s)

### (c) Calculation of heap area size

The size of the area of heap memory used is the sum of the areas allocated by memory management library functions (calloc, malloc, realloc, and new) in the C/C++ program. However, each time a memory management library function is called, either four bytes (with `cpu=H8SXN`, `cpu=H8SXM`, `cpu=H8SXA` and `ptr16` option, `cpu=H8SXX` and `ptr16` option, `cpu=2600n`, `cpu=2000n`, `cpu=300hn`, or `cpu=300` specified) or eight bytes (with `cpu=H8SXX` without `ptr16` option, `cpu=H8SXA` without `ptr16` option, `cpu=2600a`, `cpu=2000a`, or `cpu=300ha` specified) are used for management purposes; the actual area used must be calculated including the sizes of these management areas added.

The compiler manages the heap area in units of a memory size specified by the user (`_sbrk_size`). The method for specifying `_sbrk_size` is described in section 9.2.2 (5), C/C++ library function initial settings (`_INITLIB`). The heap area to be reserved (HEAPSIZE) should be calculated as follows.

$$\text{HEAPSIZE} = \text{\_sbrk\_size} \times n \ (n \geq 1)$$

(size of area allocated by memory management library functions) + management area  
size ≤ HEAPSIZE

I/O library functions use memory management library functions for internal processing.  
The size of memory allocated during I/O operations is:

With `cpu=H8SXN`, `H8SXM`, `H8SXA` (with `ptr16` option), `H8SXX` (with `ptr` option), `2600n`, `2000n`, `300hn`, `300` specified, 514 bytes x (maximum number of files open simultaneously)

With `cpu=H8SXA` (without `ptr16` option), `H8SXX` (without `ptr16` option), `2600a`, `2000a`, `300ha` specified, 516 bytes x (maximum number of files open simultaneously)

### Caution

Memory areas released using the free function or delete operator (C++) in the memory management library functions are reused by memory management library functions to secure memory; but if allocation is repeated, it is possible that requests for large memory areas cannot be satisfied, even when there is sufficient free memory available, due to the fact that free memory is broken up into smaller fragments. In order to avoid such occurrences, large memory areas should be secured immediately after the start of program execution whenever possible. In addition, the sizes of data areas which are freed and reused should be made uniform as much as possible.

#### (d) Dynamic memory area allocation

Dynamic areas are allocated in RAM.

The location for allocation of stack memory is determined by setting the uppermost address of the stack section to the SP (stack pointer) in the reset routine on program startup.

By using `__entry` (or `#pragma entry`) and `#pragma stacksize`, the C/C++ compiler automatically creates the stack area (S section) and outputs the SP initial setting code in the reset program.

The location for heap memory is determined by the initial settings for low-level interface routines (`sbrk`).

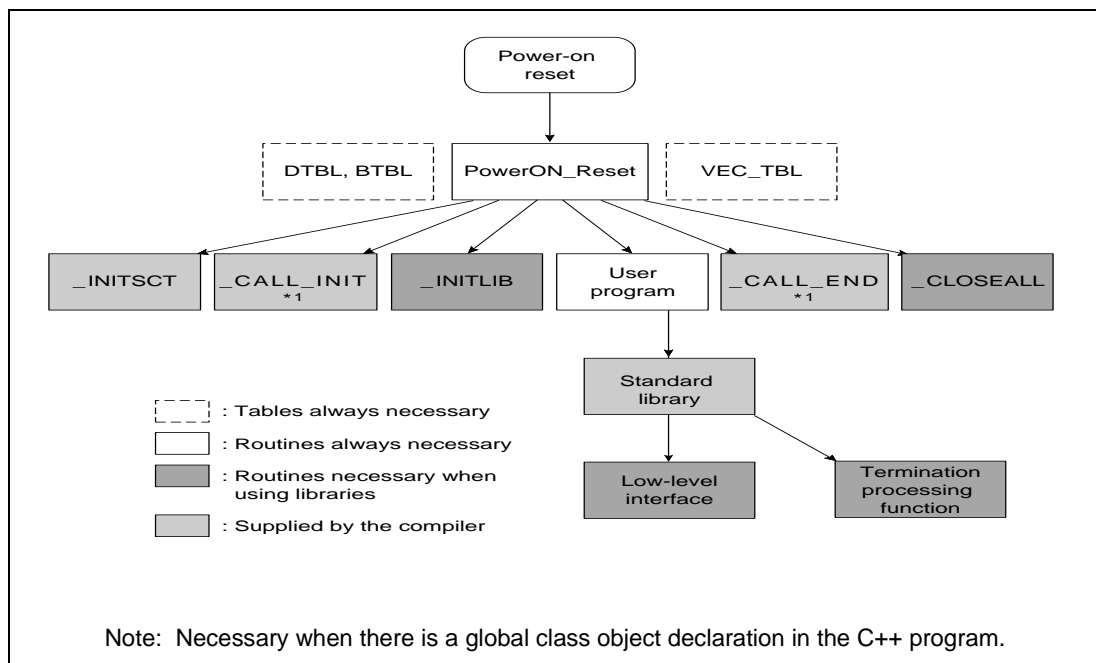
Details of each of these appear in section 9.2.2 (2), Initial settings (PowerON\_Reset), and section 9.2.2 (7), Low-level interface routines, respectively.



## 9.2.2 Execution Environment Settings

Here processing to prepare the environment for program execution is explained. 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.

Figure 9.6 shows an example of the structure of a program.



**Figure 9.6 Example of Program Structure**

The contents of each of the routines are as follows.

- Vector table (VEC\_TBL)  
Sets the vector table such that the register initial settings program (PowerON\_Reset) is started up at power-on reset.
- Initial settings (PowerON\_Reset)  
After initial register values are set, calls the initial setting routines in sequence.
- Section initialization tables (DTBL, BTBL)  
Uses the section address operator to set the leading and ending addresses for the section used in the section initialization routine.

- Section initialization (`_INITSCT`)<sup>\*1</sup>  
Initializes to zero any static variable areas (uninitialized data areas) for which no initial values are set. Also copies initial values of initialized data areas from ROM to RAM.
- Global class object initialization processing (`_CALL_INIT`)<sup>\*1\*2</sup>  
Calls the constructors for globally declared class objects.
- Global class object postprocessing (`_CALL_END`)<sup>\*1\*2</sup>  
After execution of the main function, calls the destructors for global class objects.
- C/C++ library function initial settings (`_INITLIB`)  
When using C/C++ library functions, performs initial settings for those functions requiring it.
- Close files (`_CLOSEALL`)  
Closes all open files.
- Low-level interface routines  
Routines providing an interface between the user system and library functions which are necessary when standard I/O (`stdio.h`, `ios`, `streambuf`, `istream`, `ostream`) and memory management libraries (`stdlib.h`, `new`) are used.
- Termination processing routine<sup>\*3</sup>  
Processing for terminating the program.

Notes \*1: Provided as a standard library. Include `<_h_c_lib.h>` to use `_INITSCT`, `_CALL_INIT` or `_CALL_END`

\*2: Required processing when there is a declaration of a global class object in a C++ program.

\*3: When using the C library functions `exit`, `atexit`, or `abort` to terminate a program, these functions must be created as appropriate to the user system.

When using the C library macro `assert`, the `abort` function must always be created.

Below the method for processing according to the above description is explained.

### (1) Vector table settings (VEC\_TBL)

In order to have the initial settings function PowerON\_Reset called when the system is reset at power-on, the address for the PowerON\_Reset function must be set at address 0 of the vector table.

When using interrupt processing and indirect function calls in the user system, the interrupt vectors and address table must be set appropriately.

The vector table is automatically generated by the compiler when the vect parameter is specified using the `__entry` (or `#pragma entry`), `__interrupt` (or `#pragma interrupt`), or `__indirect` (or `#pragma indirect`) extended functions of the C/C++ compiler. A code example is shown below.

```
__entry(vect=0) void PowerON_Reset(void) //PowerON_Reset function address set at address 0
{
    :
}
__interrupt(vect=3) void INT_NMI(void) //INT_NMI function address set at vector number 3
{
    :
}
__indirect(vect=4) char f //f function address set at vector number 4
{
    :
}
```

## (2) Initial settings (PowerON\_Reset)

The initial settings functions set the initial values of the stack pointer (SP) and of the condition code register (CCR) and other registers, and calls the section initialization routine (\_INITSCT) before calling the main function. When a global class object exists in a C++ program, the \_CALL\_INIT and \_CALL\_END functions, which call initialization/termination processing functions in sequence, are called before and after the main function call.

The compiler automatically generates code to set SP when \_\_entry (or #pragma entry) is used. The initial setting for the condition code register is set using an embedded function (set\_imask\_ccr etc.).

\_INITSCT and the \_CALL\_INIT and \_CALL\_END functions are provided as standard library functions. To use this function, include <\_h\_c\_lib.h>.

When using a C/C++ library function, \_INITLIB, which initializes library settings, and \_CLOSEALL, which performs processing to close files, shall be called.

A code example is shown below.

```

#include <machine.h> // Include <machine.h>
#include <_h_c_lib.h> // Include <_h_c_lib.h>
#pragma stacksize 0x200 // Set the size of section S (the stack)

extern void PowerON_Reset(void);
extern void main(void);

#ifdef __cplusplus
extern "C" {
#endif
extern void _INITLIB(void);
extern void _CLOSEALL(void);
#ifdef __cplusplus
}
#endif

__entry(vect=0) void PowerON_Reset(void)
{
    // Set SP to the uppermost address of section S
    set_vbr(0x0); // Make the initial setting of VBR for H8SX if necessary
    set_imask_ccr(1); // Mask interrupt
    _INITSCT(); // Call section initialization routine

#ifdef __cplusplus
    _CALL_INIT(); // Called when there is a global class object of C++
#endif

    _INITLIB(); // Call library initial setting function
    set_imask_ccr(0); // Release interrupt mask
    main(); // Call function to close files
    _CLOSEALL();

#ifdef __cplusplus
    _CALL_END(); // Called when there is a global class object of C++
#endif

    sleep();
}

```

### (3) Tables for section initialization (DTBL, BTBL)

The section initialization routine (`_INITSCT`) initializes any uninitialized data sections to zero, and copies initialization data in for initialized data sections in ROM to RAM. Here the starting and ending addresses of sections which is read by the `_INITSCT` function are set in the table for section initialization using the section address operator.

Section names in the section initialization table are declared, using `C$BSEC` for uninitialized data areas, and `C$DSEC` for initialized data areas.

A code example is shown below.

```
#ifdef __ABS16__          // Section name is C$DSEC.
#pragma abs16 section $DSEC
#else
#pragma section $DSEC
#endif
static const struct DSEC{
    void * rom_s;          // Start address member of the initialization data section in ROM
    void * rom_e;          // End address member of the initialization data section in ROM
    void * ram_s;          // Start address member of initialization data section in RAM
}DTBL[]={
    { _sectop ("D"), _secend ("D"), _sectop ("R") },
    { _sectop ("ABS8D"), _secend ("ABS8D"), _sectop ("ABS8R") },
    { _sectop ("ABS16D"), _secend ("ABS16D"), _sectop ("ABS16R") }
};

#ifdef __ABS16__          // Section name is C$BSEC.
#pragma abs16 section $BSEC
#else
#pragma section $BSEC
#endif
static const struct BSEC{
    void * b_s;            // Start address member of uninitialized data section
    void * b_e;            // End address member of uninitialized data section
}BTBL[]={
    { _sectop ("B"), _secend ("B") },
    { _sectop ("ABS8B"), _secend ("ABS8B") },
    { _sectop ("ABS16B"), _secend ("ABS16B") }
};
#ifdef __ABS16__
#pragma abs16 section
#else
#pragma section
#endif
#endif
```

**Note:** Be sure to compile the above program as a C language program, i.e., either make the file extension “c” or specify the **lang=c** option. If the program is compiled as a C++ program (i.e., either the file extension is “cpp”, “cc” or “cp”, or the **lang=cpp** option is specified), the table for section initialization will be deleted as an unused static data by the compiler and the program will be wrong.

The section initialization routine (`_INITSTCT`), provided as the standard library, operates similarly to the program shown below.

```
static const struct DSEC{                // Initialization table struct for D defined in previous example
    void * rom_s;                        // Start address member of the initialization data section in ROM
    void * rom_e;                        // End address member of the initialization data section in ROM
    void * ram_s;                        // Start address member of initialization data section in RAM
};

static const struct BSEC{                // Initialization table struct for B defined in previous example
    void * b_s;                          // Start address member of uninitialized data section
    void * b_e;                          // End address member of uninitialized data section
};

static void clearblock(void *b_top, void *b_end);
static void copyblock (void *d_top, void *d_end, void *r_top);

#ifdef __cplusplus
extern "C"                               // Linked to C
#endif
void _INITSTCT(void)                     // Section initialization routine
{
    const struct BSEC *btbl;             // Initialization table structure for section B
    const struct DSEC *dtbl;             // Initialization table structure for section D
                                          // Initializes the uninitialized data section
    for( btbl = __sectop ("C$BSEC");
        btbl <(struct BSEC *)__sectend ("C$BSEC"); btbl++)
        clearblock( btbl->b_s, btbl->b_e );
                                          // Initializes the initialized data section
                                          // Copies the initialized data from ROM to RAM
    for( dtbl = __sectop ("C$DSEC");
        dtbl <(struct DSEC *)__sectend ("C$DSEC"); dtbl++)
        copyblock( dtbl->rom_s, dtbl->rom_e, dtbl->ram_s );
}

static void clearblock(void *b_top, void *b_end)
{
    // Initializes the uninitialized data section by 0
    char *p;
    for( p=b_top; p<(char *)b_end; p++)
        *p = 0;
}

static void copyblock(void *d_top, void *d_end, void *r_top)
{
    // Copies the initialized data from ROM to RAM
    char *p, *q;
    for( p=r_top, q=d_top; q<(char *)d_end; p++, q++)
        *p = *q;
}
```

#### (4) C++ global class object initial settings (\_CALL\_INIT)

The \_CALL\_INIT function calls a constructor of the class object that has been globally declared in C++. Although this function is provided in the library header file of <\_h\_c\_lib,h>. An example is shown below to show the behavior.

```
extern "C" void _CALL_INIT(void);

typedef void (**FPP)(void);          // Function-pointer type

extern "C" void _CALL_INIT(void)
{
    // Global class object initialization routine
    FPP top = (FPP)_ _sectop("C$INIT");
    FPP end = (FPP)_ _secend("C$INIT");

    while (top < end)
        (*top++)();                  // Calls a constructor
}
```

#### (5) C/C++ library function initial settings (\_INITLIB)

Here, the method for setting initial values for C/C++ library functions is explained.

In order to set only those values which are necessary for the functions that are actually used, please refer to the following guidelines.

- When using the stdio.h, ios, streambuf, istream, or ostream functions or the assert macro, the standard I/O initial setting (\_INIT\_IOLIB) is necessary.
- When an initial setting is required in the created low-level interface routine, the initial setting (\_INIT\_LOWLEVEL) in accordance with the specifications of the low-level interface routine is necessary.
- When using the rand function or the 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. FILE-type data is shown in figure 9.7.



```

#include <stdio.h>
#include <stdlib.h>
#define IOSTREAM 3
const size_t _sbrk_size = 520;      // Specify minimum size to be reserved for heap area
// If omitted: _sbrk_size=1032 in advanced without ptr16 option, or maximum without ptr16 option
//           _sbrk_size=1028 in normal, middle, advanced with ptr16 option, maximum with ptr16 option, or 300
const int _nfiles = IOSTREAM;      // Specify number of I/O files (20 if omitted)
struct _iobuf _iob[IOSTREAM];
unsigned char sml_buf[IOSTREAM];
extern char *_slp_ptr;

#ifdef __cplusplus
extern "C" {
#endif
void _INITLIB (void)
{
    _INIT_LOWLEVEL();                // Set initial values for low-level interface routines
    _INIT_IOLIB();                  // Set initial values for I/O library
    _INIT_OTHERLIB();               // Set initial values for rand function, strtok function
}

void _INIT_LOWLEVEL (void)
{
    //Set necessary initial values for low-level library
}

void _INIT_IOLIB(void)
{
    FILE *fp;
    for( fp = _iob; fp < _iob + _nfiles; fp++ ) // Set initial values for FILE-type data
    {
        fp->_bufptr = NULL;
        fp->_bufcnt = 0;
        fp->_buflen = 0;
        fp->_bufbase = NULL;
        fp->_ioflag1 = 0;
        fp->_ioflag2 = 0;
        fp->_iofd = 0;
    }

    if(freopen("stdin1", "r", stdin)== NULL)    // Open standard I/O file
        stdin->_ioflag1 = 0xff; // Forbid file access if open fails
    stdin->_ioflag1 |= _IOUNBUF;                // Set without data buffering*2
    if(freopen("stdout1", "w", stdout)== NULL) // Open standard I/O file
        stdout->_ioflag1 = 0xff; // Forbid file access if open fails
    stdout->_ioflag1 |= _IOUNBUF;               // Set without data buffering*2
    if(freopen("stderr1", "w", stderr)== NULL) // Open standard error file
        stderr->_ioflag1 = 0xff; // Forbid file access if open fails
    stderr->_ioflag1 |= _IOUNBUF;               // Set without data buffering*2
}

void _INIT_OTHERLIB(void)
{
    srand(1);                                // Set initial value if using rand function
    _slp_ptr=NULL;                            // Set initial value if using strtok function
}
#ifdef __cplusplus
}
#endif

```

- Notes: 1. Specify the filename for the standard I/O file. This name is used in the low-level interface routine "open".
2. In the case of a console or other dialog-based device, a flag is set to prevent the use of buffering.

```
// File-type data declaration in C language

struct_iobuf{
    unsigned char _bufptr;           // Pointer to buffer
    long          _bufcnt;           // Buffer counter
    unsigned char _bufbase;          // Buffer base pointer
    long          _buflen;           // Buffer length
    char          _ioflag1;          // I/O flag
    char          _ioflag2;          // I/O flag
    char          _iofd;             // I/O flag
}iob[_nfiles];
```

**Figure 9.7 FILE-Type Data**

#### (6) Closing files (\_CLOSEALL)

Normally, output to files is held in a buffer area in memory, and only when the buffer becomes full is the data actually written to the external recording device. Hence if a file is not closed properly, it is possible that data output to a file may not actually be written to the external recording device.

In the case of a program intended for embedded use, normally the program is not terminated. However, if the main function is terminated due to a program error or for some other reason, any open files must all be closed.

This processing closes all the files that are open at the time of termination of the main function. An example of a program to close all the open files is shown below.

```
#include <stdio.h>

#ifdef _cplusplus
extern "C"
#endif
void _CLOSEALL(void)
{
    int i;

    for( i=0; i < _nfiles; i++ )

        if( _iob[i]._ioflag1 & (_IOREAD | _IOWRITE | _IORW ) )
            fclose( &_iob[i] );    // Close the file
}
```

## (7) 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 created. Table 9.3 lists the low-level interface routines used by C library functions.

**Table 9.3 List of Low-Level Interface Routines**

Name	Description
open	Opens file
close	Closes file
read	Reads from files
write	Writes to files
lseek	Sets the read/write position in a file
sbrk	Secures area in memory
error_addr*	Obtains errno address
wait_sem*	Waits and acquires semaphore
signal_sem*	Releases semaphore

Note: Necessary when using a reentrant library.

Initialization necessary for low-level interface routines must be performed on program startup. This initialization should be performed using the `_INIT_LOWERLEVEL` function described in section 9.2.2 (5), C/C++ library function initial settings (`_INITLIB`).

Below, after explaining the basic approach to low-level I/O, the specifications for each interface routine are described.

### Caution

The function names `open`, `close`, `read`, `write`, `lseek`, and `sbrk` are reserved words for low-level interface routine. They should not be used in user programs.

#### (a) 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 to actual files in a one-to-one correspondence for management. These integers are called file numbers.

In the `open` routine, a file number is provided for an input filename. The `open` routine must set the following information such that this number can be used for file input and output.

- The device type of the file (console, printer, disk file, etc.) (In the cases of special devices such as consoles or printers, special filenames must be set by the system and identified in the `open` routine.)
- When using file buffering, information such as the buffer position and size

- In the case of a disk file, the byte offset from the start of the file to the position for reading or writing

Based on the information set using the open routine, all subsequent I/O (read and write routines) and read/write positioning (lseek routine) is performed.

When output buffering is being used, the close routine should be executed to kick out the contents of the buffer to the actual file, so that the data area set by the open routine can be reused.

#### (b) Specifications of low-level interface routines

In this section, specifications for creation of low-level interface routines are described. For each routine, the interface for calling the routine, its operation, and any important 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. When declared in a C++ program, extern "C" should be added.

<b>(Routine name)</b>	Concise explanations
-----------------------	----------------------

Description	(A summary of the routine operations is given)
-------------	--

Return value	Normal: (The meaning of the return value on normal termination is explained)
--------------	--

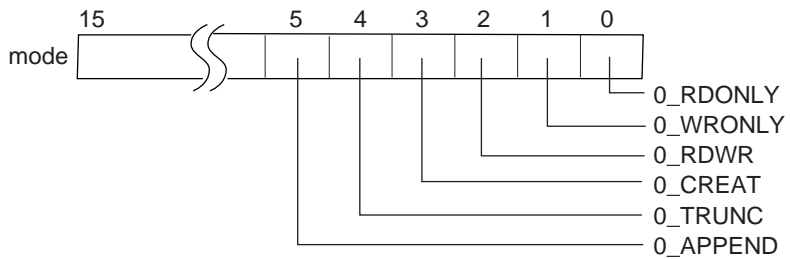
	Error: (The return value when an error occurs is given)
--	---

Parameters	(Name)	(Meaning)
	(The name of the parameter appearing in the interface)	(The meaning of the value passed as an parameter)

<b>int open(char *name, int mode, int flg)</b>	Opens file
--	------------

Description	Prepares for operations on the file corresponding to the filename of the first parameter. In the open routine, the file instance (console, printer, disk file, etc.) must be determined in order to enable reading or writing at a later time. The file instance must be accessed 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 meaning of each bit of this parameter is as follows.



**Table 9.4 Explanation of Bits in Parameter "mode" of the File 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 and if a file with the filename given does not exist, it is created
O_TRUNC (bit 4)	When this bit is 1 and if a file with the given filename 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 file beginning 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) is returned to subsequently read, write, lseek, and close routines. The correspondence between file numbers and the actual files must be managed by low-level interface routines. If the open operation fails, -1 is returned.

Return value	Normal:	The file number for the successfully opened file
	Error:	-1
Parameters	name:	Filename of the file
	mode:	Specifies the type of processing when the file is opened
	flg:	Specifies processing when the file is opened (always 0777)

## **int close(int fileno)**

Closes file

Description	<p>The file number obtained using the open routine is passed as an 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 kicked out to the actual file.</p> <p>When the file is closed successfully, 0 is returned; if the close operation fails, 1 is returned.</p>	
Return value	Normal:	0
	Error:	-1
Parameters	fileno: File number of the file to be closed	

## **int read(int fileno, char \*buf, unsigned int count)**

Reads data

Description	<p>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).</p> <p>When the end of the file is reached, only a number of bytes equal to or fewer than count bytes can be read.</p> <p>The position for file reading/writing advances by the number of bytes read.</p> <p>When reading is performed successfully, the actual number of bytes read is returned; if the read operation fails, -1 is returned.</p>	
Return value	Normal:	Actual number of bytes read
	Error:	-1
Parameters	fileno	File number of the file to be read
	buf	Memory area in which to store read data
	count	Number of bytes to read

**int write(int fileno, char \*buf, unsigned int count)**

Writes data

Description	<p>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).</p> <p>If the device (disk etc.) of the file to be written is full, only a number of bytes smaller than the 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 is judged to be full and an error (-1) is returned.</p> <p>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.</p>	
Return value	Normal:	Actual number of bytes written
	Error:	-1
Parameters	fileno	File number of the file to which data is to be written
	buf	Memory area containing data for writing
	count	Number of bytes to write

**long lseek(int fileno, long offset, int base)**

Set position in a file

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 another 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 located correctly, the new position for reading/writing is returned as an offset from the file beginning; when the operation is not successful, -1 is returned.

Return value	Normal:	The new position for file reading/writing, as an offset in bytes from the file beginning
	Error:	-1

Parameters	fileno	File number of the target file
	offset	Position for reading/writing, as an offset (in bytes)
	base	Starting-point of the offset

**char \*sbrk(size\_t size)** Allocates memory areas

Description	The size of the memory area to be allocated is passed as a parameter.	
	When calling the sbrk routine continuously, 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 is returned; if unsuccessful, (char *) -1 is returned.	
Return value	Normal:	Start address of allocated memory
	Error:	(char *) -1
Parameters	size	Size of area to be allocated

**int \*errno\_addr(void)** Acquires errno address

Description	Returns the address of the error number of the current task.	
	This routine is necessary when using a standard library, which was created by the standard library configuration tool with the <b>reent</b> option specified.	
Return value	Address of the error number of the current task	

**int wait\_sem (int semnum)** Allocates semaphore

Description	Waits and acquires the semaphore specified by semnum.	
	When semaphore has been allocated normally, 1 is returned. Otherwise, 0 is returned. This routine is necessary to use a standard library which was created by the standard library configuration tool with the <b>reent</b> option specified.	
Return value	Normal:	1
	Error:	0



Parameter	semnum	Semaphore ID
-----------	--------	--------------

<b>int signal_sem (int semnum)</b>	Releases semaphore
------------------------------------	--------------------

Description	Releases the semaphore specified by semnum.  When semaphore has been released normally, 1 is returned. Otherwise, 0 is returned. This routine is necessary to use a standard library which was created by the standard library configuration tool with the <b>reent</b> option specified.
-------------	---

Return value	Normal: 1 Error: 0
--------------	-----------------------

Parameter	semnum	Semaphore ID
-----------	--------	--------------

### (c) Example of creation of a low-level interface routine

```
/* **** */
/*                               lowsrc.c:                               */
/* - - - - - */
/*      H8S, H8/300 Series Simulator/Debugger Interface Routine      */
/*      - Only standard I/O files (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 FLMAX  3           /* Maximum number of files       */

/* file flag */
#define O_RDONLY 0x0001    /* Read only                      */
#define O_WRONLY 0x0002    /* Write only                     */
#define O_RDWR  0x0004    /* Both read and Write           */

/* special character code */
#define CR 0x0d            /* Carriage return               */
#define LF 0x0a            /* Line feed                     */

/* Area size managed by sbrk */
#if _ _DATA_ADDRESS_SIZE_ _== 4
#define HEAPSIZE 2064
#else
#define HEAPSIZE 2056
#endif

/* **** */
/* Declaration of reference function                                */
/* Reference to assembly program in which the simulator debugger inputs or */
/* outputs characters to the console                                */
/* **** */

extern void charput(char);    /* One character output          */
extern char charget(void);    /* One character input           */
```

```

/*****
/* Definition of static variables: */
/* Definition of static variables used in low-level interface routines */
/*****

char flmod[FLMAX];          /* Open file mode specification area */
static union {
    short dummy;           /* Dummy for 2-byte boundary */
    char heap[HEAPSIZE];    /* Declaration of the area managed by sbrk */
} heap_area;
static char *brk=(char *)&heap_area; /* End address of area assigned by sbrk */

/*****
/* open: file open */
/* Return value: File number (Pass) */
/* -1 (Failure) */
/*****

extern open(char name,          /* File name */
            int mode,           /* File mode */
            int flg)           /* Unused */
{
    /* Checks mode depending on file name and returns file numbers */

    if(strcmp(name,"stdin")==0){ /* Standard input file */
        if((mode&O_RDONLY)==0)
            return -1;
        flmod[STDIN]=mode;
        return STDIN;
    }

    else if(strcmp(name,"stdout")==0){ /* Standard output file */
        if((mode&O_WRONLY)==0)
            return -1;
        flmod[STDOUT]=mode;
        return STDOUT;
    }

    else if(strcmp(name,"stderr")==0){ /* Standard error file */
        if((mode&O_WRONLY)==0)
            return -1;
        flmod[STDERR]=mode;
        return STDERR;
    }

    else
        return -1; /* Error */
}

```

```

/*****
/*   close: File close
/*       Return value: 0   (Pass)
/*                   -1 (Failure)
*****/

extern close(int fileno)          /* File number */
{
    if(fileno<FLMIN || FLMAX<=fileno) /* File number range check */
        return -1;

    flmod[fileno]=0;                /* File mode reset */
    return 0;
}

/*****
/* read: Data read
/*       Return value: Number of read characters (Pass)
/*                   -1 (Failure)
*****/

extern read(int  fileno,          /* File number */
            char buf,            /* Destination buffer address */
            int  count)          /* Number of read characters */
{
    int i;
    /* Checks mode according to file no. and stores each character in buffer */

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

```

```

/*****
/* write: Data write
/*      Return value: Number of write characters (Pass)
/*      -1 (Failure)
*****/

extern write(int  fileno,          /* File number
      char buf,          /* Destination buffer address
      int  count)        /* Number of write characters

{
    int  i;
    char c;

    /* Checks mode according to file no. and outputs each character
    if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR){
        for(i=count; i>0; i--){
            c=*buf++;
            charputc;
        }
        return count;
    }
    else
        return -1;
}

/*****
/* lseek: Definition of file read/write position
/*      Return value: Offset from the top of file read/write position (Pass)
/*      -1 (Failure)
/*      (lseek is not supported in the console input/output)
*****/

extern long lseek(int  fileno,          /* File number
      long offset,          /* Read/write position
      int  base)            /* Origin of offset

{
    return -1L;
}

```

```

/*****
/*      sbrk: Data write
/*      Return value: Start address of the assigned area (Pass)
/*      -1 (Failure)
*****/

extern char sbrk(size_t size)          /* Assigned area size */
{
    char *p ;

    if(brk+size>heap_area.heap+HEAPSIZE) /* Empty area size */
        return (char *)-1 ;
    p=brk ;                               /* Area assignment */
    brk += size ;                         /* End address update */
    return p ;
}

```

```

;- - - - -
;                                lowlvl.nor                                |
;- - - - -
;      H8S, H8/300 Series Simulator/Debugger Interface Routine          |
;                                -Input/output one character-            |
;- - - - -
;      H8SX, H8S/2600, H8S/2000, H8/300H normal mode                    |
; (cpu=H8SXN, 2600n, 2000n, 300hn)                                     |
;- - - - -

        .CPU          2600N          ; or H8SXN, 2000N, 300HN
        .EXPORT      _charput
        .EXPORT      _charget

SIM_IO:  .EQU          H'00FE          ; Defines TRAP_ADDRESS

        .SECTION      P, CODE, ALIGN=2

;- - - - -
; _charput: One character output                                         |
;      C program interface: charput(char)                               |
;- - - - -

_charput:
        MOV.B         R0L, @IO_BUF      ; Specifies parameter in buffer
        MOV.W         #H'0102, R0       ; Specifies parameter and function code
        MOV.W         #LWORD IO_BUF, R1
        MOV.W         R1, @PARM         ; Specifies I/O buffer address
        MOV.W         #LWORD PARM, R1   ; Specifies parameter block address
        JSR           @SIM_IO
        RTS

```

```

;- -----
;  _charget: One character input                                     |
;      C program interface:char charget(void)                       |
;- -----

_charget:
    MOV.W      #H'0101,R0      ; Specifies parameter and function code
    MOV.W      #LWORD IO_BUF,R1
    MOV.W      R1,@PARM        ; Specifies I/O buffer address
    MOV.W      #LWORD PARM,R1  ; Specifies parameter block address
    JSR        @SIM_IO
    MOV.B      @IO_BUF,R0L
    RTS

;- -----
;  I/O buffer definition                                           |
;- -----

        .SECTION      B,DATA,ALIGN=2

PARM:    .RES.W      1          ; Parameter block area
IO_BUF:  .RES.B      1          ; I/O buffer area

        .END

```



```

;- - - - -
;                               lowlvl.adv                               |
;- - - - -
;   H8S, H8/300 Series Simulator/Debugger Interface Routine           |
;                               -Input/output one character-           |
;- - - - -
;   H8SX, H8S/2600, H8S/2000, and H8/300H in advanced mode (20|24-bit address) |
;   (cpu=H8SXA:20|24, 2600a:20|24, 2000a:20|24, 300ha)               |
;- - - - -
;   .CPU           2600A           ; or H8SXA, 2000A, 300HA
;   .EXPORT        _charput
;   .EXPORT        _charget

SIM_IO:  .EQU          H'01FE          ; Defines TRAP_ADDRESS

        .SECTION      P,CODE,ALIGN=2

;- - - - -
;   _charput: One character output                                     |
;   C program interface: charput(char)                               |
;- - - - -

_charput:
        MOV.B        R0L,@IO_BUF      ; Specifies parameter in buffer
        MOV.W        #H'0112,R0       ; Specifies parameter and function code
        MOV.L        #IO_BUF,ER1
        MOV.L        ER1,@PARM        ; Specifies I/O buffer address
        MOV.L        #PARM,ER1        ; Specifies parameter block address
        JSR          @SIM_IO
        RTS

```

```

;- - - - -
; _charget: One character input
; C program interface: char charget(void)
;- - - - -

_charget:
    MOV.W    #H'0111,R0    ; Specifies parameter and function code
    MOV.L    #IO_BUF,ER1
    MOV.L    ER1,@PARM     ; Specifies I/O buffer address
    MOV.L    #PARM,ER1     ; Specifies parameter block address
    JSR      @SIM_IO
    MOV.B    @IO_BUF,R0L
    RTS

;- - - - -
; I/O buffer definition
;- - - - -

        .SECTION          B,DATA,ALIGN=2

PARM:    .RES.L           1          ; Parameter block area
IO_BUF:  .RES.B           1          ; I/O buffer area

        .END

```

```

;- - - - -
;                                     lowlvl.mid
;- - - - -
;   H8S, H8/300 Series Simulator/Debugger Interface Routine
;                                     Input/Output one character
;- - - - -
;   H8SX Middle mode, H8SX Advanced/Maximum mode(16-bit data address)
; (cpu=H8SXM, cpu=H8SXA ptr16, cpu=H8SXX ptr16)
;- - - - -

        .CPU           H8SXM
        .EXPORT        _charput
        .EXPORT        _charget
SIM_IO:  .EQU          H'01FE      ; Specify TRAP_ADDRESS

        .SECTION       P, CODE, ALIGN=2

;- - - - -
;   _charput: One character output
;           C program interface: charput(char)
;- - - - -

_charput:
        MOV.B          R0L,@IO_BUF      ; Set parameter to buffer
        MOV.W          #H'0102,R0      ; Set parameter and function code
        MOV.W          #LWORD IO_BUF,R1
        MOV.W          R1,@PARM        ; Set I/O buffer address
        MOV.W          #LWORD PARM,R1  ; Set parameter block address
        JSR            @SIM_IO
        RTS

```

```

;- - - - -
; _charget: One character input
; C program interface: char charget(void)
;- - - - -

_charget:
    MOV.W    #H'0101,R0      ; Set parameter and function code
    MOV.W    #LWORD IO_BUF,R1
    MOV.W    R1,@PARM        ; Set I/O buffer address
    MOV.W    #LWORD PARM,R1  ; Set parameter block address
    JSR      @SIM_IO
    MOV.B    @IO_BUF,R0L
    RTS

;- - - - -
;                                     Definition of I/O buffer
;- - - - -

        .SECTION      B,DATA,ALIGN=2

PARM:   .RES.W        1          ; Parameter block area
IO_BUF: .RES.B        1          ; I/O buffer area

        .END

```

```

;- - - - -
;
;                               lowlvl.max
;- - - - -
;   H8S, H8/300 Series Simulator/Debugger Interface Routine
;                               Input/Output one character
;- - - - -
;   H8SX Maximum mode, H8SX,H8S/2600,H8S/2000 Advanced mode(28|32-bit address)|
;   (cpu=H8SXX, H8SXA:28|32, 2600a:28|32, 2000a:28|32)
;- - - - -

        .CPU           H8SXX
        .EXPORT        _charput
        .EXPORT        _charget
SIM_IO:  .EQU          H'01FE          ; Specify TRAP_ADDRESS

        .SECTION       P, CODE, ALIGN=2

;- - - - -
;   _charput: One character output
;   C program interface: charput(char)
;- - - - -

_charput:
        MOV.B          R0L,@IO_BUF    ; Set parameter to buffer
        MOV.W          #H'0122,R0     ; Set parameter and function code
        MOV.L          IO_BUF,ER1
        MOV.L          ER1,@PARM      ; Set I/O buffer address
        MOV.L          #PARM,ER1      ; Set parameter block address
        JSR            @SIM_IO
        RTS

```

```

;- - - - -
; _charget: One character input
; C program interface: char charget(void)
;- - - - -

_charget:
    MOV.W        #H'0121,R0        ; Set parameter and function code
    MOV.L        IO_BUF,ER1
    MOV.L        ER1,@PARM          ; Set I/O buffer address
    MOV.L        #PARM,ER1          ; Set parameter block address
    JSR          @SIM_IO
    MOV.B        @IO_BUF,R0L
    RTS

;- - - - -
;                                     Definition of I/O buffer
;- - - - -

        .SECTION        B,DATA,ALIGN=2

PARM:   .RES.L        1                ; Parameter block area
IO_BUF: .RES.B        1                ; I/O buffer area

        .END

```

```

;- - - - -
;                               lowlvl.reg                               |
;- - - - -
;   H8S, H8/300 Series Simulator/Debugger Interface Routine           |
;                               -Input/output one character-           |
;- - - - -
;                               H8/300 (cpu=300)                       |
;- - - - -

        .CPU          300
        .EXPORT      _charput
        .EXPORT      _charget

SIM_IO:  .EQU          H'00FE          ; Defines TRAP_ADDRESS

        .SECTION      P, CODE, ALIGN=2

;- - - - -
;   _charput: One character output                                     |
;   C program interface: charput(char)                               |
;- - - - -

_charput:
        MOV.B         R0L, @IO_BUF      ; Specifies parameter in buffer
        MOV.W         #H'0102, R0       ; Specifies parameter and function code
        MOV.W         #IO_BUF, R1
        MOV.W         R1, @PARM         ; Specifies I/O buffer address
        MOV.W         #PARM, R1         ; Specifies parameter block address
        JSR           @SIM_IO
        RTS

```

```

;- - - - -
; _charget: One character input
; C program interface: char charget(void)
;- - - - -

_charget:
    MOV.W    #H'0101,R0      ; Specifies parameter and function code
    MOV.W    #IO_BUF,R1
    MOV.W    R1,@PARM        ; Specifies I/O buffer address
    MOV.W    #PARM,R1        ; Specifies parameter block address
    JSR      @SIM_IO
    MOV.B    @IO_BUF,R0L
    RTS

;- - - - -
; I/O buffer definition
;- - - - -

        .SECTION      B,DATA,ALIGN=2

PARM:   .RES.W    1          ; Parameter block area
IO_BUF: .RES.B    1          ; I/O buffer area

        .END

```



(d) Example of low-level interface routines for reentrant library

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

When an error is returned from the **wait\_sem** function or **signal\_sem** function, set **errno** as follows to return from the library function.

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

When an interrupt with a priority level higher than the current level is generated after semaphores have been allocated, dead locks will occur if semaphores are allocated again. Therefore, be careful for processes that share resources because they might be nested by interrupts.

```

#define MALLOC_SEM      1      /* Semaphore No. for malloc */
#define STRTOK_SEM      2      /* Semaphore No. for strtok */
#define FILE_TBL_SEM    3      /* Semaphore No. for _iob */
#define SEMSIZE          4
#define TRUE             1
#define FALSE            0
#define OK                1
#define NG                0

extern int *errno_addr(void);
extern int wait_sem(int);
extern int signal_sem(int);

int sem_errno;
int force_fail_signal_sem = FALSE;
static int semaphore[SEMSIZE];

/*****
/*          errno_addr:Acquisition of errno address          */
/*          Return address: errno address                      */
*****/
int *errno_addr(void)
{
    /* Return the errno address of the current task */
    return (&sem_errno);
}

```

```

/*****
/*      wait_sem: Acquires the specified number of semaphores      */
/*
/*          Return value: OK(=1) (Normal)                          */
/*
/*          NG(=0) (Error)                                         */
*****/
int wait_sem(int semnum) /* Semaphore ID */
{
    if((0 <= semnum) && (semnum < SEMSIZE)) {
        if(semaphore[semnum] == FALSE) {
            semaphore[semnum] = TRUE;
            return(OK);
        }
    }
    return(NG);
}

/*****
/*      signal_sem: Releases the specified number of semaphores    */
/*
/*          Return value: OK(=1) (Normal)                          */
/*
/*          NG(=0) (Error)                                         */
*****/
int signal_sem(int semnum) /* Semaphore ID */
{
    if(!force_fail_signal_sem) {
        if((0 <= semnum) && (semnum < SEMSIZE)) {
            if( semaphore[semnum] == TRUE ) {
                semaphore[semnum] = FALSE;
                return(OK);
            }
        }
    }
    return(NG);
}

```

## (8) Termination processing routines

### (a) Example of creation of a routine for termination processing registration and execution (atexit)

The method for creation 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 an 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.

```
#include <stdlib.h>
typedef void *atexit_t ;

int _atexit_count=0 ;

atexit_t (*_atexit_buf[32])(void) ;

#ifdef __cplusplus
extern "C"
#endif
atexit_t atexit(atexit_t (*f)(void))
{
    int i;

    for(i=0; i<_atexit_count ; i++)    // Check whether the function has
        if(_atexit_buf[i]==f)        // already been registered
            return NULL ;
    if (_atexit_count==32)            // Check whether the limit for
        return NULL ;                // registered functions is exceeded
    else {
        _atexit_buf[_atexit_count++]=f ; // Register the function address
        return f ;
    }
}
```

(b) Example of creation of a routine for program termination (exit)

The method for creation 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 creating a termination procedure according to the specifications of the user system.

The exit function performs termination processing for a program according to the termination code for the program passed as a parameter, and returns to the environment in which the program was started. Here the termination code is set to an external variable, and execution returned to the environment saved by the setjmp function immediately before the main function was called. In order to return to the environment prior to program execution, the following callmain function should be created, and instead of calling the function main from the PowerON\_Reset initial setting function, the callmain function should be called.

A program example is shown below.

```

#include <setjmp.h>
#include <stddef.h>

typedef void * atexit_t ;
extern int _atexit_count ;

extern atexit_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 registered by
        (*_atexit_buf[i])(); // the atexit function
    _CLOSEALL(); // Close all open functions
    longjmp(_init_env, 1) ; // Return to the environment saved by setjmp

#ifdef _ _cplusplus
extern "C"
#endif
void callmain(void)
{
    // Save the current environment using setjmp,
    // call the main function

    if(!setjmp(_init_env))
        _exit_code=main(); // On returning from the exit function,
    // terminate processing
}

```

(c) Example of creation of an abnormal termination (abort) routine

On abnormal termination, execute the abnormal termination procedure according to the user system.

When using the C++ program, the abort function is called in the following cases:

- When correct exception processing was not performed
- When a pure virtual function is called
- When `dynamic_cast` failed
- When `typeid` failed
- When information could not be obtained when the class array was deleted
- When contradiction occurred when destructor call information for class object was called

Below is an example of a program which outputs a message to the standard output device, then closes all files and begins an endless 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 aborted !!\n"); //Output message
    _CLOSEALL();                  //Close all files
    while(1)                      //Begin endless loop
}
```

## 9.3 Linking C/C++ Programs and Assembly Programs

Through its support for #pragma statements, keywords and other extended features as well as functions, this compiler provides all functions necessary for programs of embedded use equipment via the C and C++ languages.

However, in cases where there are strict demands on performance, such as when hardware timing is required or when the size of memory is limited, it may be necessary to write code in assembly language integrated into the C/C++ program.

Keep the following in mind when joining C/C++ programs and assembly programs.

- Method for mutual referencing of external names
- Interface for function calling

### 9.3.1 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 static storage classes (C/C++ programs)
- Variable names declared as extern storage classes (C/C++ programs)
- Function names not specified as static memory classes (C programs)
- Non-member, non-inline function names not specified as static memory classes (C++ programs)
- Non-inline member function names (C++ programs)
- Static data member names (C++ programs)



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

In assembly programs, the .EXPORT directive 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 (defines the name)

```
.EXPORT    _a,_b
.SECTION D,DATA,ALIGN=2
_a: .DATA.W 1
_b: .DATA.W 1
.END
```

C/C++ program (references the name)

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

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

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

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

C/C++ program (defines the name)

```
char a,b;
```

Assembly program (references the name)

```
.IMPORT    _a,_b
.SECTION P,CODE,ALIGN=2
MOV.B     @_a,R5L
MOV.B     R5L,@_b
RTS
.END
```

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

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

C++ program (defines the name)

```
extern "C"
int f(int a)
{
    ...
}
```

Assembly program (references the name)

```
.IMPORT    _f
.SECTION P,CODE,ALIGN=2
:
JSR        @_f
:
.END
```

### 9.3.2 Function Calling Interface

When calling functions in both directions between a C/C++ program and an assembly program, four collections of rules, explained below, must be followed on the assembly program side.

- Rules concerning the stack pointer
- Rules concerning allocation and release of stack frames
- Rules concerning registers
- Rules concerning settings and referencing parameters and return values

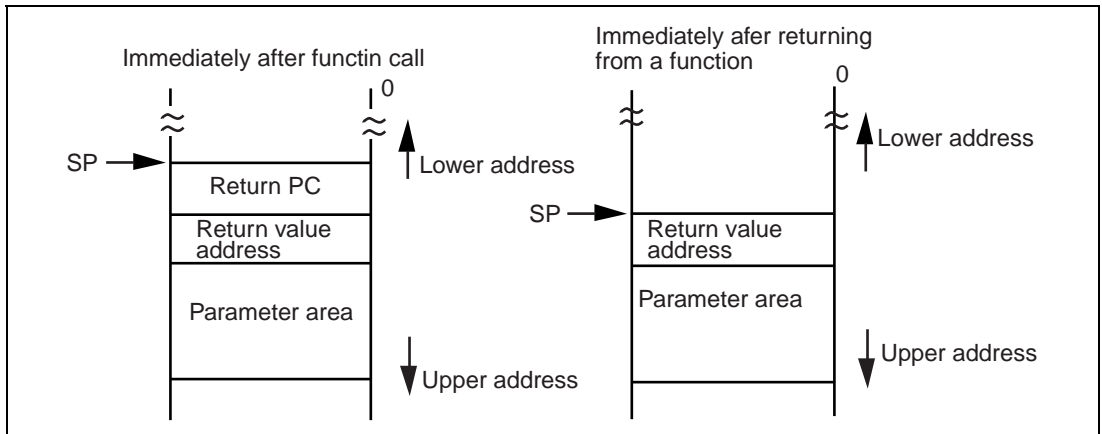
#### (1) Rules concerning the stack pointer

No valid data must be stored in the stack area below (in the direction toward address 0) the address indicated by the stack pointer. The data may become corrupted by interrupt processing.

#### (2) Rules concerning allocation and release of stack frames

At the time of a function call (after execution of a JSR or BSR instruction), the stack pointer points to a return PC area. The calling function allocates area above this area and sets data.

When the function returns, the return PC area is released by the called function. This is normally performed using the RTS instruction. Areas at addresses above this (return value addresses and parameter areas) are released by the calling function.



**Figure 9.8 Rule for Allocation and Release of Stack Frames**

### (3) Rules concerning registers

There are registers which guarantee a value to remain the same before and after a function call, and registers which do not. Rules for guaranteeing register values for different CPU types appear in table 9.5.

**Table 9.5 Rules for Guaranteeing Register Values Before and After Function Calls**

Type	Number of Registers for Storing Parameters	CPU Type and Registers		Important Information When Programming
		H8SX, H8S/2600, H8S/2000, H8/300H	H8/300	
Registers which do not guarantee values (caller-save)	2	ER0, ER1	R0, R1	If there is a valid value in a register when a function is called, the calling function saves the value; the called function can use the register without saving its contents
	3	ER0 to ER2	R0 to R2	
Registers which do guarantee values (callee-save)	2	ER2 to ER6	R2 to R6	The contents of the registers used within the function are saved, and are restored on return
	3	ER3 to ER6	R3 to R6	

Note: The number of registers used to store parameters can be set using the **regparam** option or `__regparam2`, `__regparam3`.

Below are specific examples of rules for guaranteeing register values, in the case of the H8S/2600 advanced mode.

(a) Calling an assembly program subroutine from a C/C++ program

Assembly program (called function)

```

        .EXPORT    _sub
        .SECTION   P, CODE, ALIGN=2
_sub:   STM.L      (ER4-ER6), @-SP
        SUB.L      #10, SP
        :
        ADD.L      #10, SP
        LDM.L      @SP+, (ER4-ER6)
        RTS
        .END

```

- } Contents of registers to be used within the function are saved by the callee
- } Function body (ER0, ER1 can be used without saving)
- } Saved register contents restored

C/C++ program (calling function)

```

#ifdef __cplusplus
extern "C"
#endif
void sub(void);
void f(void)
{
    sub();
}

```

(b) Calling a C program subroutine from an assembly program

C program (called function)

```

void sub(void)
{
    ...
}

```

Assembly program (calling function)

```

        .IMPORT    _sub
        .SECTION   P, CODE, ALIGN=2
        :
        MOV.L      ER1, @(4, SP)
        MOV.L      ER0, ER6
        JSR        @_sub
        :
        RTS
        .END

```

If there are valid values in registers ER0, ER1, they are saved by the caller to unused registers or to the stack

Function name referenced with \_ prepended

(c) Calling a C++ program subroutine from an assembly program

C++ program (called function)

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

Assembly program (calling function)

```
.IMPORT    _sub
.SECTION   P, CODE, ALIGN=2
:
MOV.L     ER1, @(4, SP)
MOV.L     ER0, ER6
JSR       @_sub
:
RTS
.END
```

If there are valid values in registers ER0, ER1, they are saved by the caller to unused registers or to the stack

Note: Functions declared using extern "C" cannot be overloaded.

(4) Rules concerning settings and referencing parameters and return values

Below the method for setting and referencing parameters and return values is explained. Rules for parameters and return values differ depending on whether, in the function declaration, the type of each parameter and of the return value has been declared explicitly or not. In order to make explicit declarations of the types of parameters and the return value, a function prototype declaration is used.

In the following explanation, first general rules for parameters and return values are described; then, assignment of parameters and the location for setting the return value are discussed.

(a) General rules for parameters and return values

- Passing parameters

The values of parameters must always be copied to the area allocated to parameters before calling the function. The calling function does not reference the area allocated to parameters after return, and so the called function can change the parameter values with no direct effect on processing by the calling function.

- Rules for type conversion:

When passing parameters or returning a value, in some cases automatic type conversions are performed. Below the rules for these type conversions are explained.

Type conversion of return values:

Return values are converted into the type returned by the function.

Type conversion of parameters for which a type is declared:

Parameters for which a type has been declared using a prototype declaration are converted to the declared type.

Type conversion of parameters for which no type is declared:

Type conversions of parameters for which no type has been declared using a prototype declaration are performed according to the following rules.

- Parameters with the char and unsigned char types are converted to the int type.
- Parameters with the float type are converted to the double type.
- All types other than the above are not converted.

Example:

(1) `long f();`

```
long f()
{
    float x;
    return x;
}
```

→ The return value is converted to long.

(2) `void p(int, ...);`

```
f()
{
    char c;
    p(1.0, c);
}
```

→ c is converted to int because no type is declared for the parameter.

→ 1.0 is converted to int because int type is declared for the parameter.

## Caution

When parameter types are not declared using a prototype declaration, if the same type is not specified by both the calling and the called function so as to ensure that the correct parameters are passed, correct operation is not guaranteed.

```
f(x)
float x;
{
    .
    .
}

main()
{
    float x;
    f(x);
}
```

Example of a case in which correct operation is not guaranteed

```
f(float x)
{
    .
    .
}

main()
{
    float x;
    f(x);
}
```

Example of a case in which correct operation is guaranteed

In the example of a case in which correct operation is not guaranteed, the parameter `x` is converted to the double type by the function `main` because function `f` has no parameter prototype declaration. On the other hand, the parameter is declared as the float type by the function `f`. Hence the parameter cannot be passed correctly. Either the parameter type should be declared using a prototype declaration, or else the parameter declaration for function `f` should be changed to the double type.

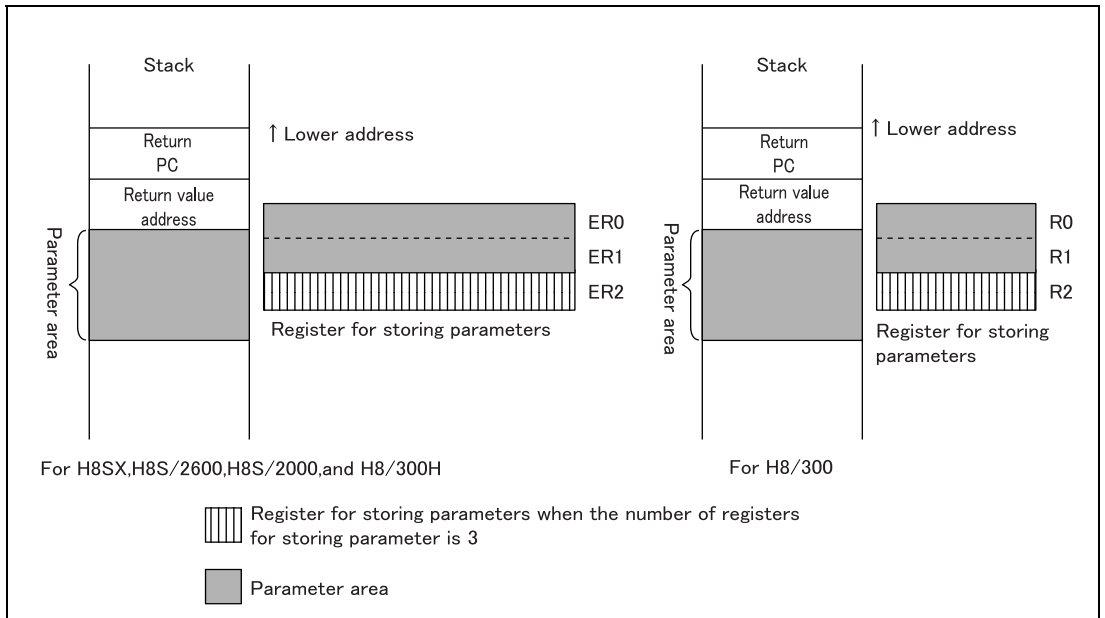
In the example of correct type specification, the parameter type is declared using a prototype declaration.

### (b) Area for allocation of parameters

Parameters are allocated to an area on the stack in some cases, and to registers in others.

Areas for allocation of parameters by object type are shown in table 9.6, and general rules for areas for allocation of parameters are indicated in figure 9.9.

The “this” pointer of nonstatic function members in C++ programs is allocated to `R0` or `ER0`.



**Figure 9.9 Memory Area for Allocation of Parameters**



**Table 9.6 General Rules for Memory for Allocation of Parameters**

CPU Type	Number of Registers for Parameter Storage	Rules for Allocation		
		Parameters for Allocation to Registers		Parameters for Allocation to the Stack
		Parameter Storage Registers	Parameter Types for Storage	
H8SX	2	ER0, ER1	char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, structures (4 bytes or less)* <sup>4</sup> , pointers, references, pointers to data members	[1] Parameter type is other than a type allocable to registers  [2] Function is declared by a prototype declaration as a function with a variable number of parameters* <sup>2</sup>  [3] Parameters which cannot be allocated to registers because of the large number of parameters
H8S/2600	3	ER0, ER1, ER2		
H8S/2000				
H8/300H				
H8/300	2	R0, R1	char, unsigned char, short, unsigned short, int, unsigned int, long* <sup>3</sup> , unsigned long* <sup>3</sup> , float* <sup>3</sup> , structures (2 bytes or less)* <sup>4</sup> , structures (4 bytes or less)* <sup>3*4</sup> , pointers, references, pointers to data members	
	3	R0, R1, R2		

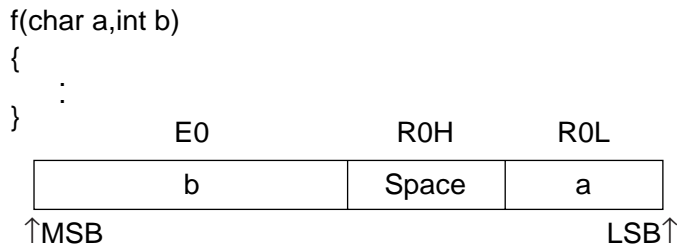
- Notes: 1. The number of registers for parameter storage can be specified using the `regparam` option or `__regparam2` and `__regparam3`.
2. When a function is declared using a prototype declaration as having a variable number of parameters, parameters in the ... part, and the parameter immediately preceding the ... part, are allocated on the stack.

Example:     `int f2(int, int, ...);`  
                  `f2(x,y,z);` → y, z are allocated to the stack

3. When the **longreg** option is specified.
4. When the **structreg** option is specified.

### (c) Parameter allocation

- Allocation of registers for parameter storage  
Allocation of registers for parameter storage is performed in the order of parameter declaration in the source program, starting from the LSB side of the lowest-numbered register. An example of allocation of registers for parameter storage appears in figure 9.10.



**Figure 9.10 Example of Allocation of Registers for Parameter Storage (H8S/2600)**

- Allocation to parameter area on the stack  
Parameters are allocated to areas on the stack for parameters in the order specified in the source program, starting from lowest addresses.

### Caution

When specifying parameters that are structures, unions or classes, 2-byte boundary alignment is used regardless of the normal byte alignment for that type, and an area with an even number of bytes is used. This is because in the H8SX, AE5, H8S, H8/300H and H8/300 series, the stack pointer changes in 2-byte units.

In section 9.3.3, Examples of Parameter Assignment, specific examples of parameter allocation for different CPU/operating modes are described.

#### (d) Location for setting return values

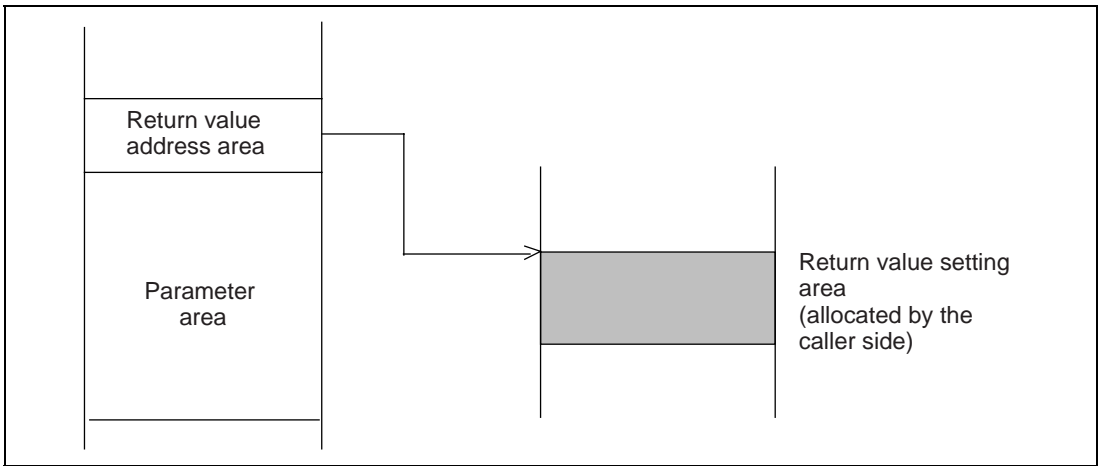
Depending on the type of the value returned by a function, the return value may be set in either a register or in memory. The relation between the return value type and the location for storage is described in table 9.7. When setting a function return value in memory, the return value is set in the area indicated by the return value address. The caller function secures an area for the return value, the area for parameters, and the area to set the address of the return value, calls the function (cf. figure 9.11).

If the return value of a function is of type void, no return value is set.

**Table 9.7 Return Value Types and Location in Memory**

Return Value Type	Location for Setting Return Value	
	H8SX, AE5, H8S/2600, H8S/2000, H8/300H	H8/300
char, unsigned char	Register (R0L)	Register (R0L)
short, unsigned short, int, unsigned int	Register (R0)	Register (R0)
Ponter to function	Register Normal mode: (R0) The other mode: (ER0)	Register (R0)
Pointer to data, reference, pointer to a data member	Register Normal/Middle mode: (R0) Advanced/Maximum mode with ptr16 option or __ptr16 keyword: (R0)* <sup>3</sup> Advanced/Maximum mode without ptr16 option and __ptr16 keyword: (ER0)	Register (R0)
long, unsigned long, float	Register (ER0)	Area for setting return values (memory) Register (R0, R1)* <sup>1</sup>
Structures of 2 bytes or less	Area for setting return values (memory) Register (R0)* <sup>2</sup>	Area for setting return values (memory) Register (R0)* <sup>2</sup>
Structures of 3 or 4 bytes	Area for setting return values (memory) Register (ER0)* <sup>2</sup>	Area for setting return values (memory) Register (R0, R1)* <sup>1*2</sup>
double, long double, structure, union, class, pointer to a function member	Area for setting return values (memory)	Area for setting return values (memory)

- Notes: 1. When the **longreg** option is specified.  
2. When the **structreg** option is specified.  
3. The ptr16 option and the \_\_ptr16 keyword are valid only with the H8SX and H8S CPU



**Figure 9.11 Area for Setting Return Values in Memory**

### 9.3.3 Examples of Parameter Assignment

(1) For the H8SX, H8S/2600, H8S/2000, H8/300H (cpu=H8SXN, cpu=H8SXM, cpu=H8SXA, cpu=H8SXX, cpu=2600a, cpu=2600n, cpu=2000a, cpu=2000n, cpu=300ha, cpu=300hn)

Example 1: Parameters of types for passing to registers are assigned, in the order of declaration, to registers ER0 and ER1\*<sup>1</sup>.

[1] int f(char, char, char);	R0L	1
:		
f(1, 2, 3);	R0H	2
:		
	R1L	3

[2] int f(int, int, int);	R0	1
:		
f(1, 2, 3);	E0	2
:		
	R1	3

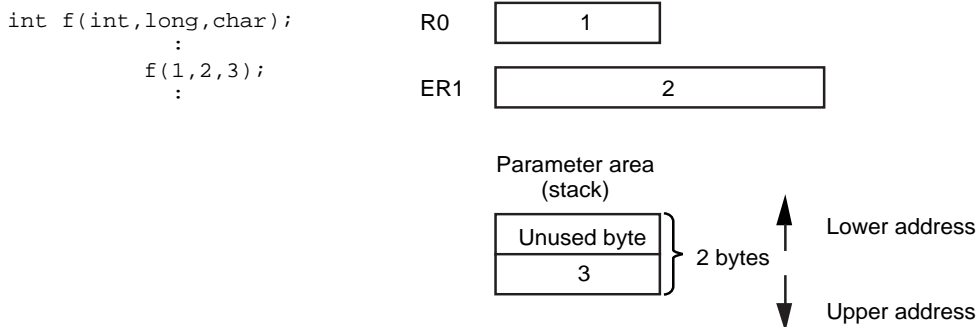
[3] int f(long, long);	ER0	1
:		
f(1, 2);	ER1	2
:		

[4] int f(char, int, int, char);	R0L	1
:		
f(1, 2, 3, 4);	E0	2
:		
	R1	3
	R0H	4

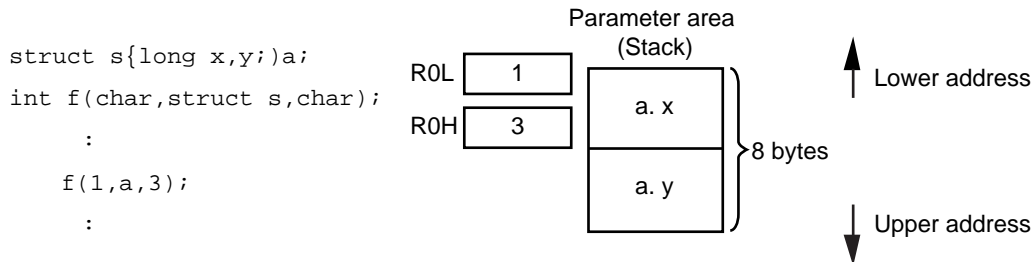
Note: When there are three registers for storing parameters, the registers are ER0, ER1, and ER2.

Example 2: Parameters which cannot be assigned to registers are assigned to the stack. When an parameter of type char is assigned to the parameter area on the stack, the lower bytes are invalid.

(Case in which there are two registers for parameter storage)

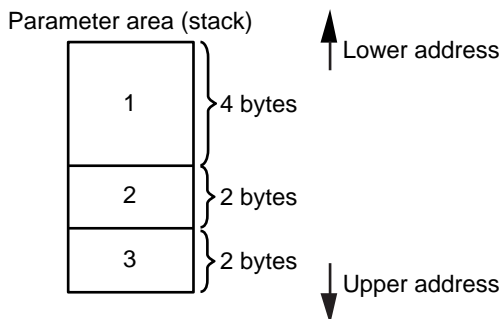


Example 3: Parameters of types which cannot be assigned to registers are assigned to the stack.

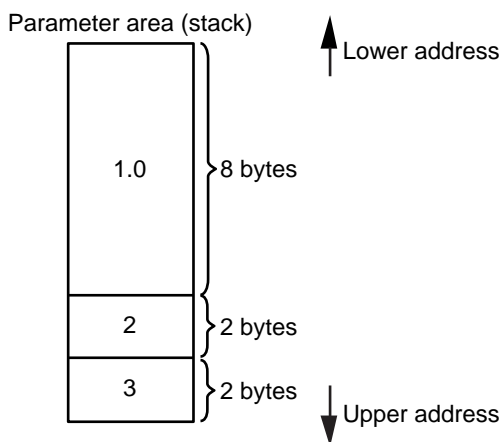


Example 4: When a function is declared as having a variable number of parameters using a prototype declaration, a parameter without a corresponding type and the immediately preceding parameter are assigned to the stack in the order of declaration.

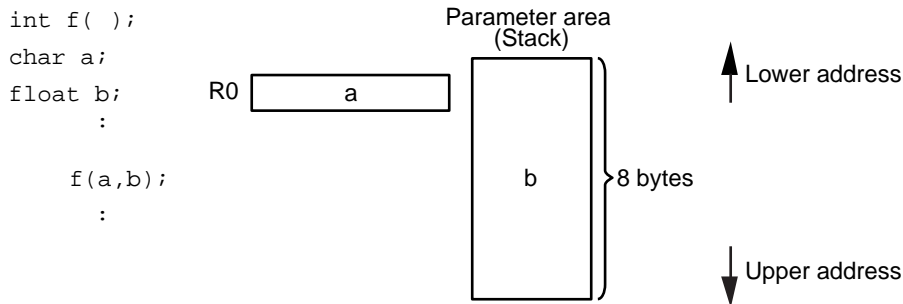
```
[1] int f(long,...);  
    :  
    f(1,2,3);  
    :
```



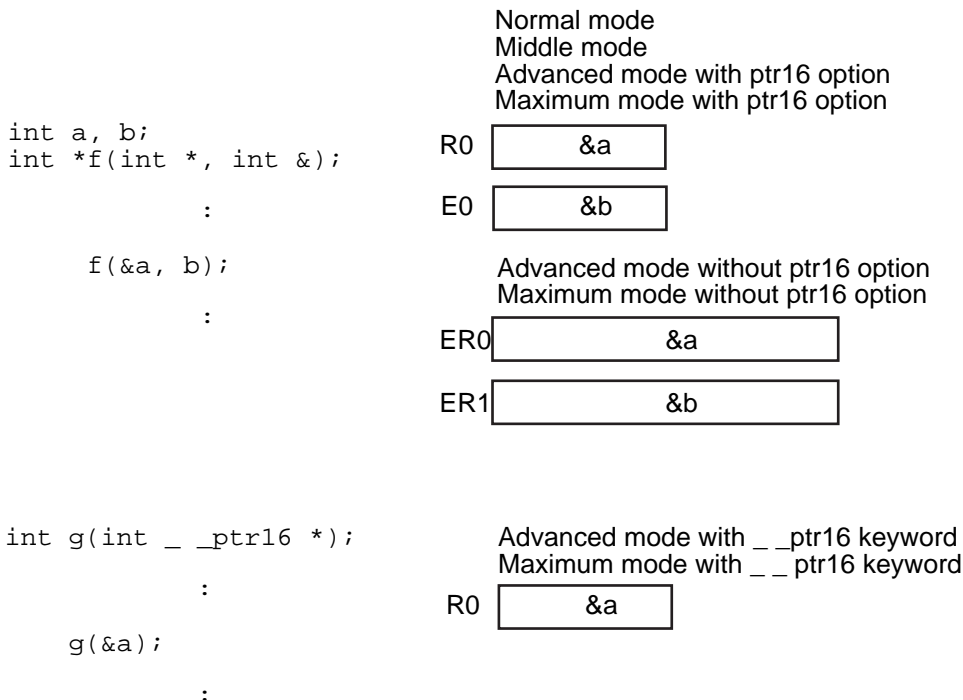
```
[2] int f(double,int,...);  
    :  
    f(1.0,2,3);  
    :
```



Example 5: When there is no prototype declaration in a C program, char is expanded to the int type, and float is expanded to the double type for passing.



Example 6: The pointer-to-data type and the reference type of C++ are assigned to 2-byte areas in normal or middle mode and in advanced or maximum mode with ptr16 option or `__ptr16` keyword, and to 4-byte areas in advanced or maximum mode without ptr16 option and `__ptr16` keyword. Note that ptr16 option and `__ptr16` keyword is effective only with H8SX and H8S.





Example 7: The return value of pointer-to-data type are assigned to 2-byte areas in normal or middle mode and in advanced or maximum mode with ptr16 option or `__ptr16` keyword, and to 4-byte areas in advanced or maximum mode without ptr16 option and `__ptr16` keyword. Note that ptr16 option and `__ptr16` keyword is effective only with H8SX and H8S.

```
int *f(void);
int *p;
```

```
:
```

```
p = f( );
```

```
:
```

Normal mode  
Middle mode  
Advanced mode with ptr16 option  
Maximum mode with ptr16 option

R0

f

Advanced mode without ptr16 option  
Maximum mode without ptr16 option

ER0

f

```
int __ptr16 *g(void);
int __ptr16 *q;
```

```
:
```

```
q = g( );
```

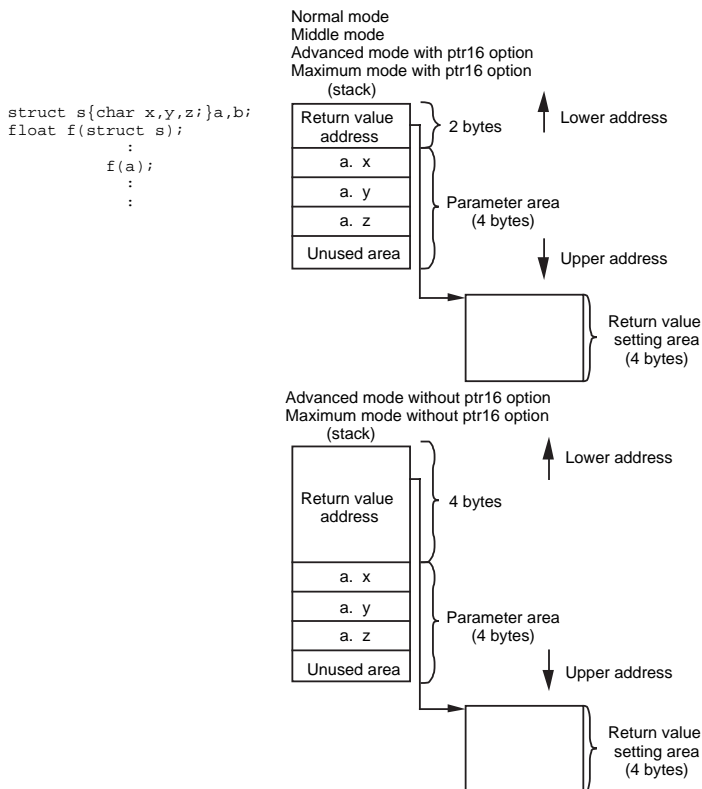
```
:
```

Advanced mode with `__ptr16` keyword  
Maximum mode with `__ptr16` keyword

R0

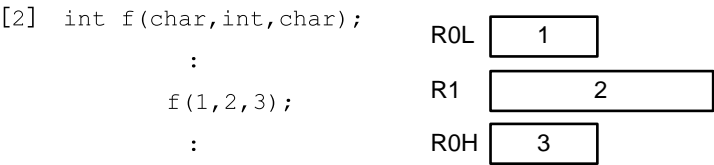
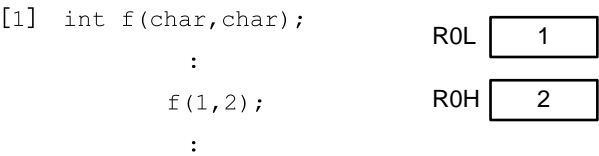
g

Example 8: When the type returned by a function exceeds 4 bytes, or when it is a structure (when structreg is not specified, or when the structure exceeds 4 bytes), the return value address is set immediately before the parameter area. Also, when a structure size is an odd number of bytes, one unused byte of memory area results.



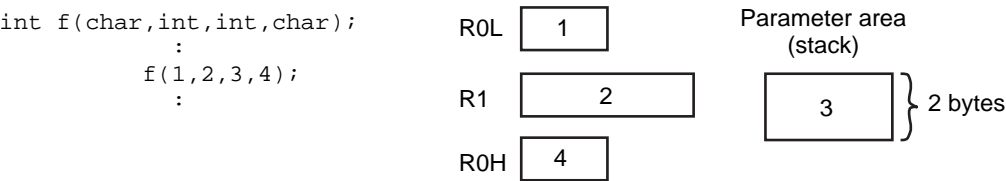
(2) For the H8/300 (cpu=300)

Example 1: Parameters of types for passing to registers are assigned, in the order of declaration, to registers R0 and R1\*<sup>1</sup>.

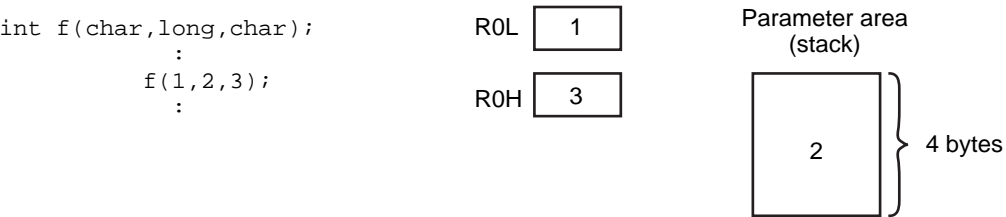


Note: When there are three registers for storing parameters, the registers are R0, R1, and R2.

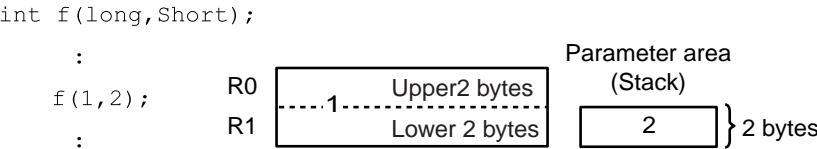
Example 2: Parameters which cannot be assigned to registers are assigned to the stack. (Case in which there are two registers for parameter storage)



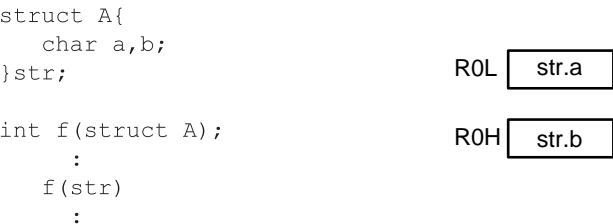
Example 3: Parameters of types which cannot be assigned to registers are assigned to the stack.



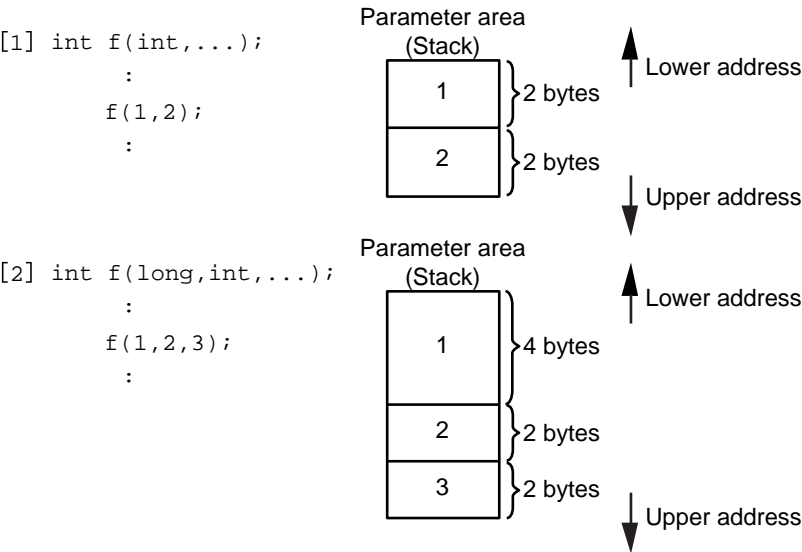
Example 4: When the **longreg** option is specified, four-byte data is assigned to registers R0 and R1.



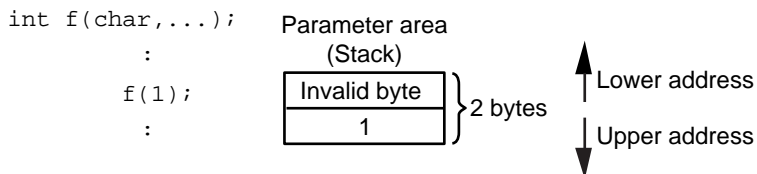
Example 5: When the **structreg** option is specified, structures of 2 bytes or less are assigned to registers.



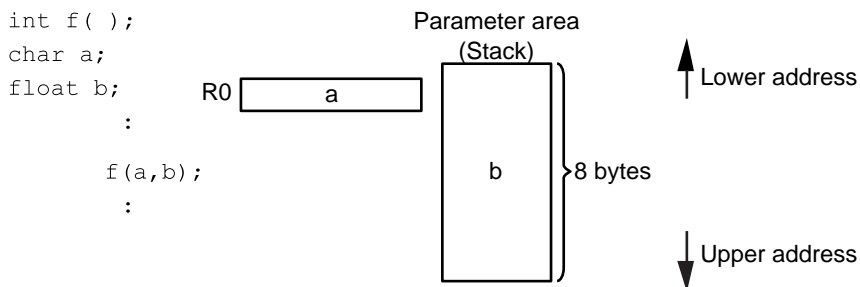
Example 6: When a function is declared as having a variable number of parameters using a prototype declaration, an parameter without a corresponding type and the immediately preceding parameter are assigned to the stack in the order of declaration.



Example 7: When an parameter of type char is assigned to the parameter area on the stack, the lower bytes are invalid.

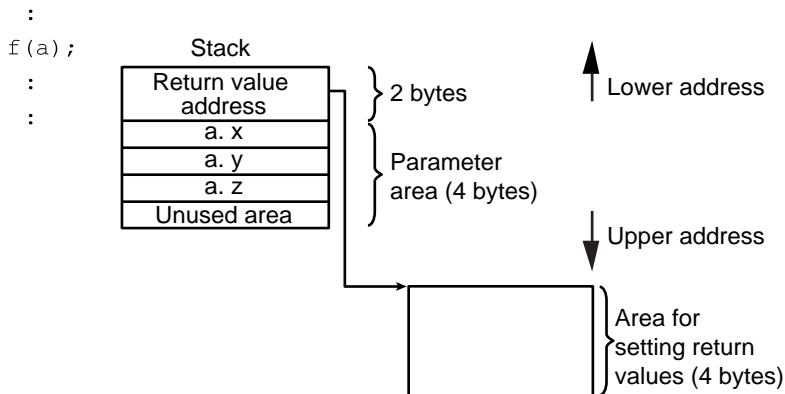


Example 8: When there is no prototype declaration in a C program, char is expanded to the int type, and float is expanded to the double type for passing.

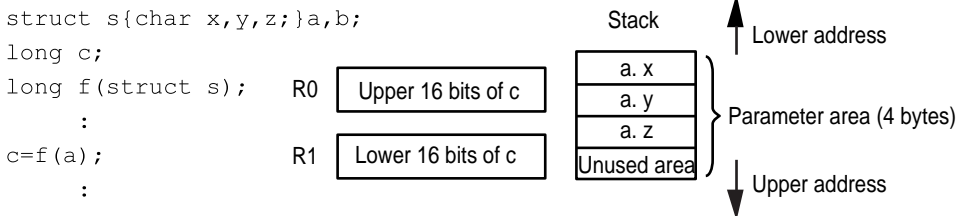


Example 9: When the type returned by a function exceeds 2 bytes, the return value address is set immediately before the parameter area. Also, when a structure size is an odd number of bytes, one unused byte of memory area results.

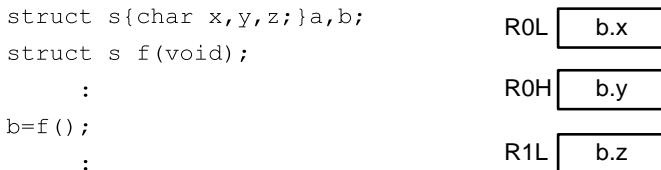
```
struct s{char x,y,z;}a,b;
float f(struct s);
```



Example 10: When the **longreg** option is specified, if the type returned by a function exceeds 2 bytes, the return value is assigned to registers R0 and R1.

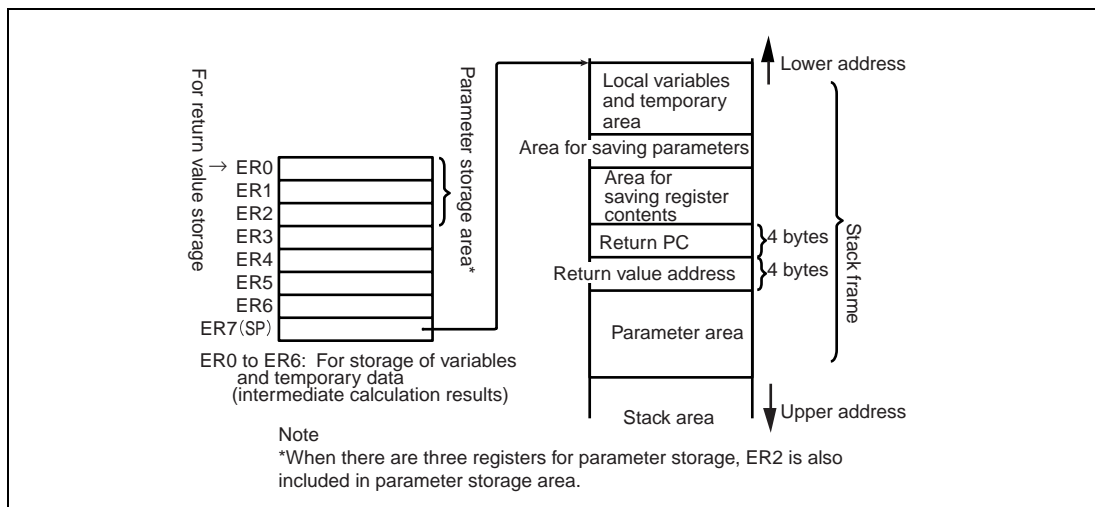


Example 11: When the **structreg** and **longreg** options are specified, if the type returned by a function is a structure of 4 bytes or less, the return value is assigned to registers R0 and R1.



### 9.3.4 Using the Registers and Stack Area

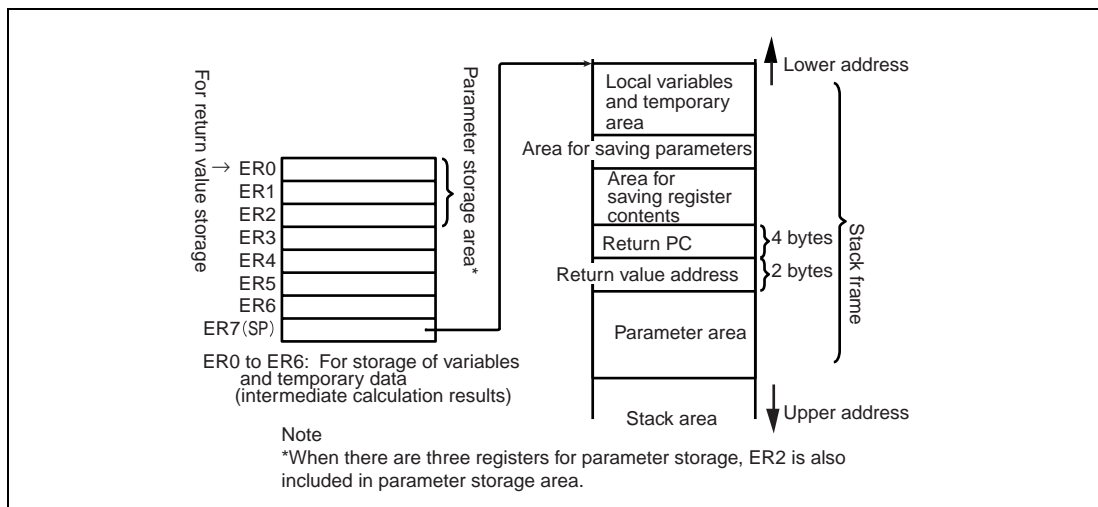
(1) For the H8SX advanced mode and maximum mode (cpu=H8SXA, cpu=H8SXX)



**Figure 9.12 Using Registers and Stack Area (cpu=H8SXA<sup>\*1</sup>, cpu=H8SXX<sup>\*1</sup>)**

Note: 1. Without the ptr16 option.

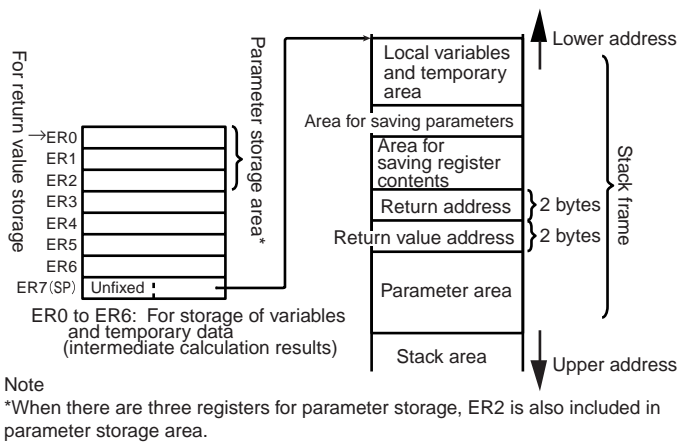
(2) For the H8SX middle mode, advanced mode with ptr16, maximum mode with ptr16 (cpu=H8SXM, cpu=H8SXA with ptr16, CPU=H8SXX with ptr16)



**Figure 9.13 Using Registers and Stack Area (cpu=H8SXM, cpu=H8SXA<sup>\*2</sup>, cpu=H8SXX<sup>\*2</sup>)**

Note: 2. With the ptr16 option.

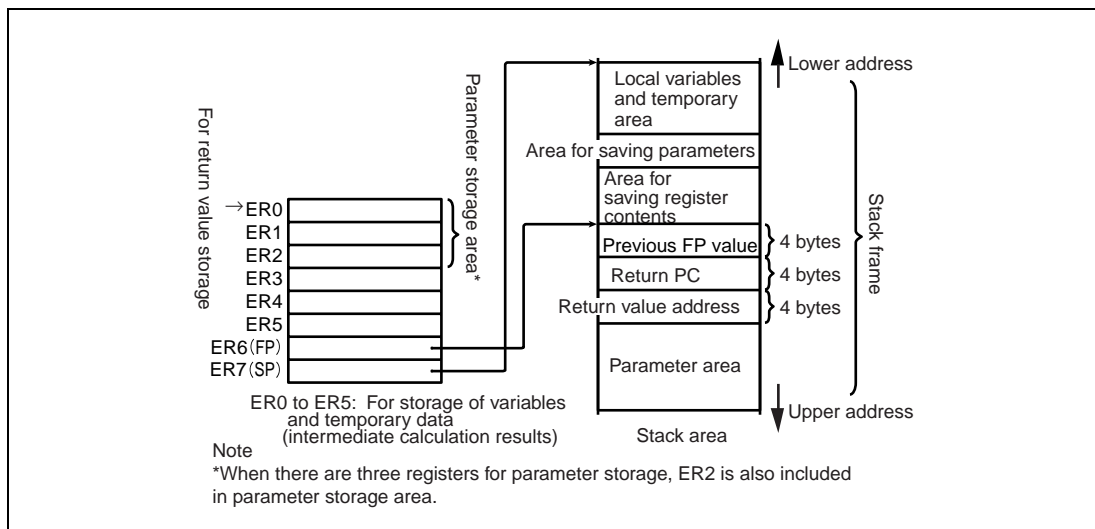
(3) For the H8SX normal mode (cpu=H8SXN)



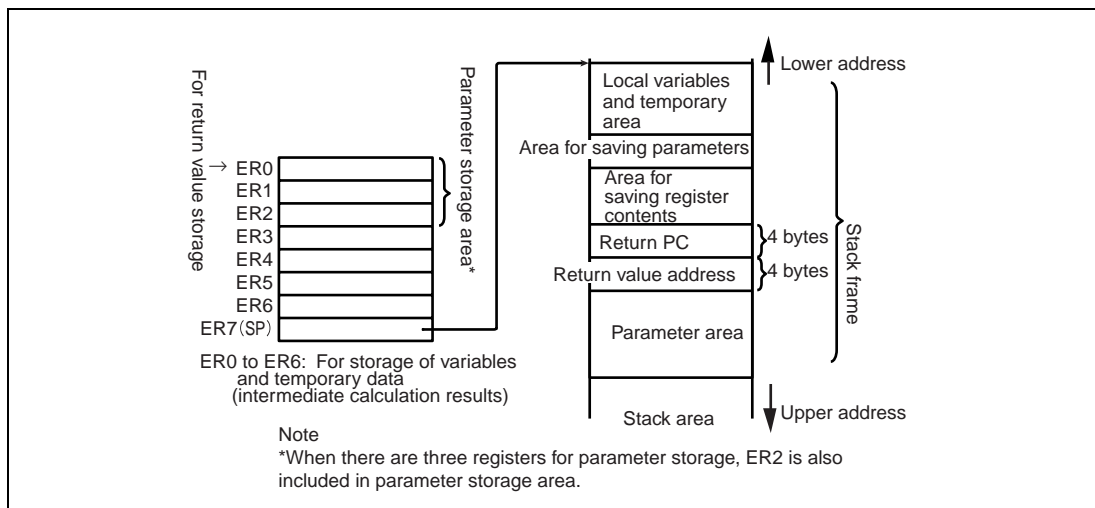
**Figure 9.14 Using Registers and Stack Area (cpu=H8SXN)**



(4) For the H8S/2600, H8S/2000 and H8/300H advanced mode (cpu=2600a, cpu=2000a, cpu=300ha)

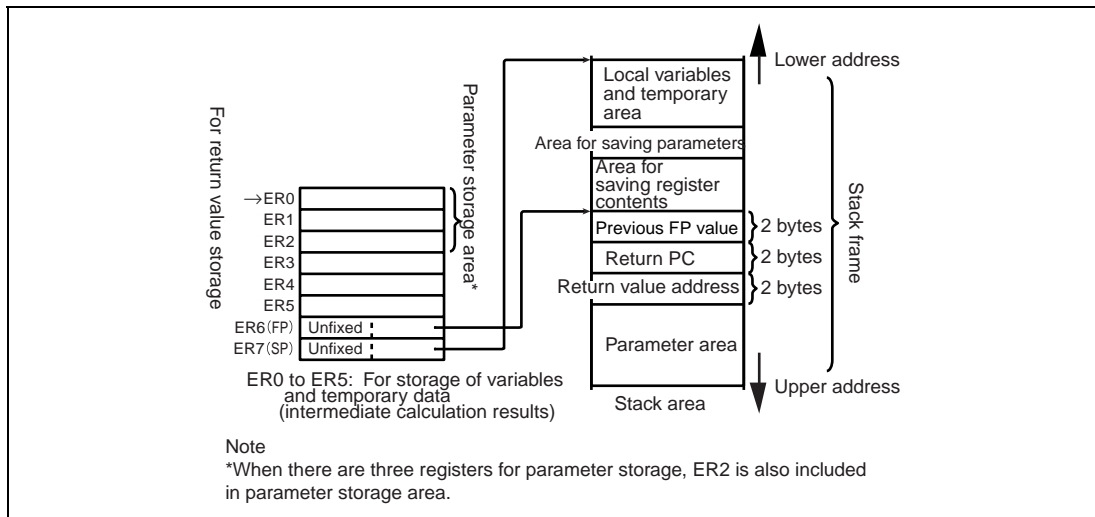


**Figure 9.15 Using Registers and Stack Area without Optimization (cpu=2600a, cpu=2000a, cpu=300ha)**

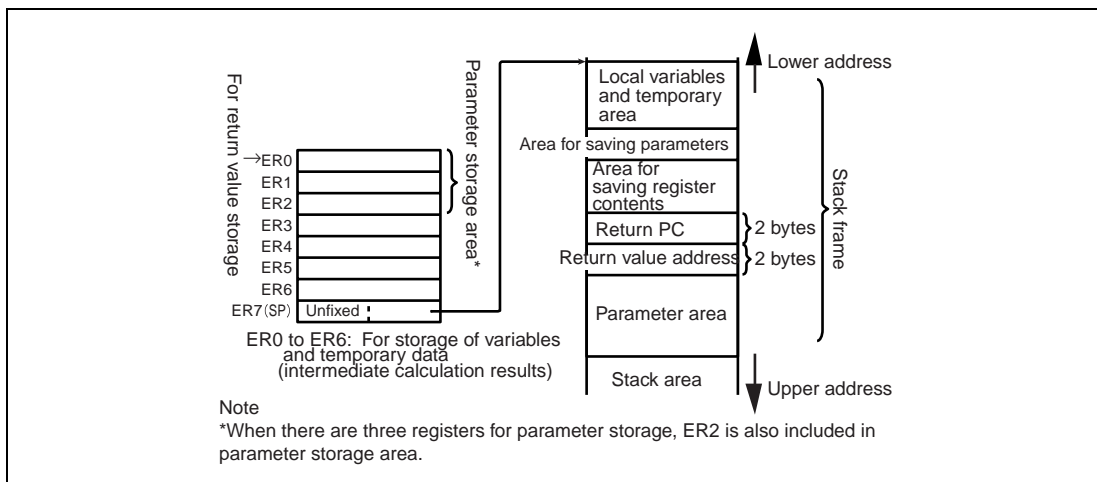


**Figure 9.16 Using Registers and Stack Area with Optimization (cpu=2600a, cpu=2000a, cpu=300ha)**

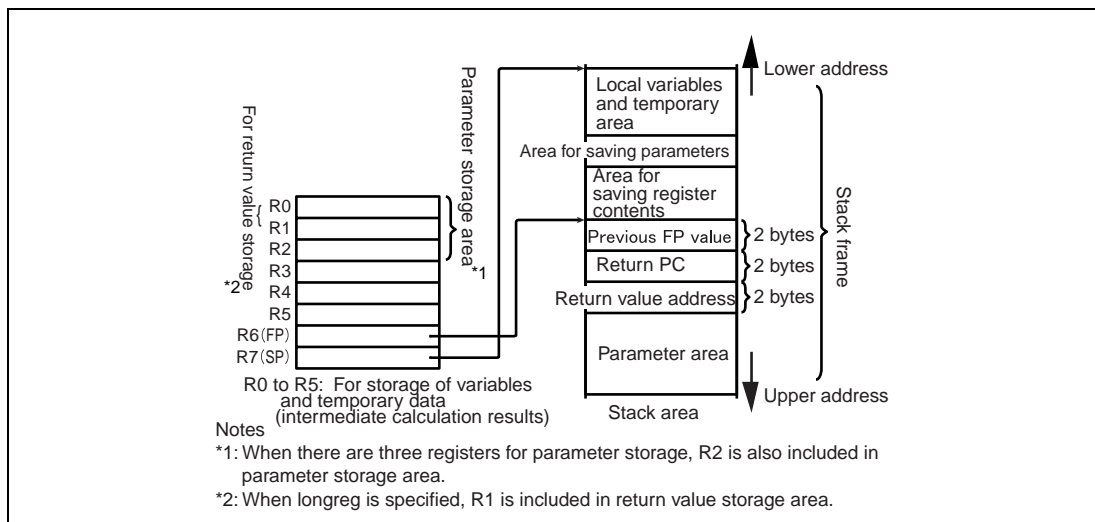
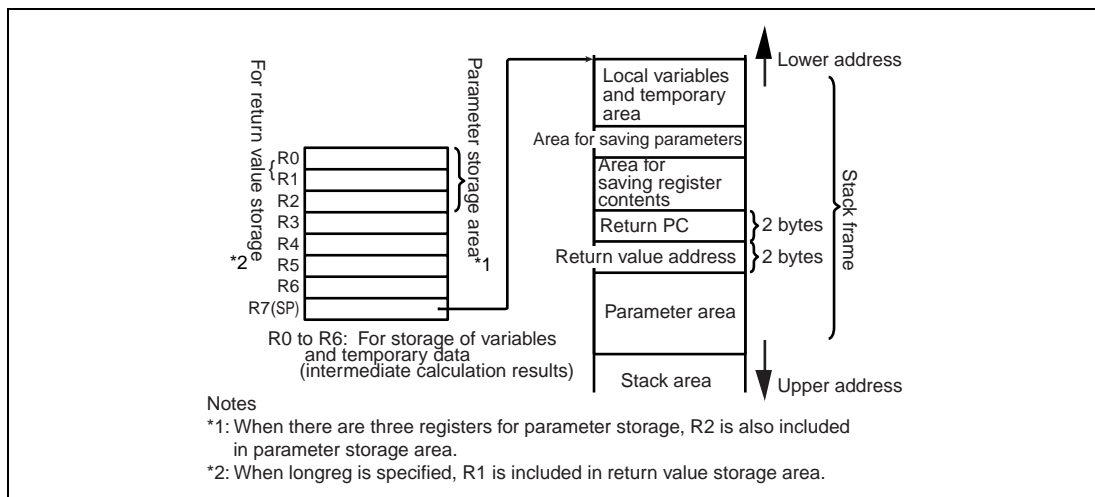
(5) For the H8S/2600, H8S/2000, and H8/300H normal mode (cpu=2600n, cpu=2000n, cpu=300hn)



**Figure 9.17 Using Registers and Stack Area without Optimization (cpu=2600n, cpu=2000n, cpu=300hn)**



**Figure 9.18 Using Registers and Stack Area with Optimization (cpu=2600n, cpu=2000n, cpu=300hn)**

**Figure 9.19 Using Registers and Stack Area without Optimization (cpu=H8/300)****Figure 9.20 Using Registers and Stack Area with Optimization (cpu=H8/300)**

## 9.4 Important Information on Program Creation

In this section, important information on writing program code for the compiler, and matters to bear in mind during development of a program from compiling through debugging, are described.

### 9.4.1 Important Information on Program Coding

#### (1) Functions taking float type parameters

Functions which declare a float type parameter should always be given a prototype declaration, or else the float type should be changed to the double type in the parameter declaration. If a function which takes a float type parameter but does not have a prototype declaration is called, correct operation is not guaranteed.

Example:

```
void f(float); -----[1]

void g(
{
    float a;
    :
    f(a);
}

void f(float x)
{
    :
}
```

The function f takes a float type parameter. Here a prototype declaration like that in [1] should always be used.

#### (2) Expressions for which order of evaluation is not specified by the C/C++ language

If an expression is used for which the order of evaluation is not stipulated by the C/C++ language, and the result of the expression changes depending on the order of evaluation, then correct operation is not guaranteed.

Example:

<code>a[i]=a[++i];</code>	The value of i on the left-hand side changes depending on whether the assignment expression on the right is evaluated first or last.
<code>sub(++i, i);</code>	The value of i of the second parameter changes depending on whether the first parameter of the function is evaluated first or last.

#### (3) Code which may be deleted through optimization

When the same variable is referenced continuously, or an expression whose result is not used is written, such code may be deleted as redundant by the compiler as part of optimization. In order to ensure constant access, the volatile keyword should be used in the declaration.

Example:

```
[1] b=a;          /* The expression on the first line may be deleted as redundant code */
    b=a;
[2] while(1)a;    /* The reference of the variable a and loop statement may be */
                  /* deleted as redundant */
```

#### (4) Overflow operations and division by zero

No error message is output even if there is an overflow operation or division by zero. However, in an operation on a single constant or a pair of constants, if there is an overflow or division by zero, an error message is output at compile time. In H8SX, however, the compiler might not detect division by zero.

Example:

```
void main(void)
{
    int ia;
    int ib;
    float fa;
    float fb;
    ib=32767;
    fb=3.4e+38f;

    /* Compiler error message is output in response to overflow or */
    /* division by zero for an operation on a constant or pair of constants */
    ia=99999999999; /* (W) Detects overflow of constant */
    fa=3.5e+40f     /* (W) Detects overflow of floating-point operation */
    ia=1/0; /* (E) Detects division by zero excluding H8SX and H8S */
    fa=1.0/0.0;     /* (W) Detects floating-point division by zero */
                  /* excluding H8SX and H8S */
    /* No error message is output in response to an overflow at runtime */
    ib=ib+32767;    /* Overflow in operation result ignored */
    fb=fb+3.4e+38f; /* Overflow in floating-point operation result ignored */
}
```

#### Caution

When the **cpuexpand** option is specified, no overflow or underflow errors are output.

(5) On the precision of mathematical library functions

The error in the **acos**(x) and **asin**(x) functions is great when  $x \approx 1$ ; care should be taken when using these functions. The error range is as follows.

Absolute error at double precision in  $\text{acos}(1.0 - \epsilon) \ 2^{-39}$  ( $\epsilon = 2^{-33}$ )

At single precision  $2^{-21}$  ( $\epsilon = 2^{-19}$ )

Absolute error at double precision in  $\text{asin}(1.0 - \epsilon) \ 2^{-39}$  ( $\epsilon = 2^{-28}$ )

At single precision  $2^{-21}$  ( $\epsilon = 2^{-16}$ )

(6) Writing to const type variables

Keep the following in mind. If a variable declared as const is converted to a type that is not const via type conversion, or if types are not consistent among files compiled separately, then the compiler cannot check for writing to a const type variable.

Examples:

```
[1] const char *p; /* The first parameter of the library function strcat is */
    :             /* a pointer to a char type, and so the area indicated */
    strcat(p, "abc"); /* by the parameter may be overwritten. */

[2] File 1
    const int i;

    File 2
    extern int i; /* The variable i is not declared as const type in File 2. */
    :           /* No error is detected against update of i. */
    i=10;
```

(7) Note on bit manipulation instructions

This compiler generates the bit manipulation instructions BSET, BCLR, BNOT, BST, and BIST. These instructions read data in byte units, and after bit manipulation write data in byte units again. On the other hand, if a write-only register is read, the CPU retrieves an undefined value, regardless of the register contents. Hence in bit manipulation instructions for a write-only register, bits other than the bit to be manipulated may change. The following is an example of bit manipulation for a write-only register.

Example:

Contents of the include file (300x.h)

```
struct S_p4ddr{
    unsigned char p7:1;
    :
    unsigned char p0:1;
};
union SS{
    unsigned char Schar;
    struct S_p4ddr Sstr;
};
#define P4DDR (*(union SS *)0xffffc5)
#define P0 0x1
```

Contents of the C source program

```
#include "300x.h"
unsigned char DDR;
// Prepare backup data for write-only
// register
void sub(void)
{
    DDR &=~P0;
    P4DDR.Schar=DDR;
}
```

## 9.4.2 Important Information on Compiling a C Program with the C++ Compiler

### (1) Function prototype declarations

Before using a function, a prototype declaration is necessary. At this time the types of parameters should also be declared.

```
extern void func1();
void g()
{
    func1(1); // error in C++
}
```

```
extern void func1(int);
void g()
{
    func1(1); // OK
}
```

### (2) Linkage of const objects

Whereas in C programs const objects are linked externally, in C++ programs they are linked internally. In addition, const objects require initial values.

```
const int cvalue1;
// error in C++
const int cvalue2=1;
// local in C++
```

```
const int cvalue1=0;
// initial value required

extern int const cvslue2=1;
// has external linkage like C
```

### (3) Substitution from void\*

In C++ programs, if explicit casting is not used, substitution into pointers to other objects (excluding pointers to functions and to members) is not possible.

```
void func(void *ptrv,int *ptri)
{
    ptri = ptrv; // error in C++
}
```

```
void func(void *ptrv,int *ptri)
{
    ptri = (int *)ptrv; // OK
}
```

### 9.4.3 Important Information on Program Development

Important information for program development, from program creation through debugging, is described below.

#### (1) Information concerning selection of the CPU/operating mode

- (a) The same CPU/operating mode should be specified at compile time and assembly time.

The CPU/operating mode specified using the **cpu** option at compile time and assembly time must always be the same. If object programs created for different CPU/operating modes are linked, operation of the object program at runtime is not guaranteed.

- (b) The same CPU type as the CPU/operating mode specified at compile time should be specified at assembly time.

When assembling an assembly program generated by the C compiler, the **cpu** option should be used to specify the same CPU type specified by the CPU/operating mode at compile time.

- (c) The same CPU type as the CPU/operating mode specified at compile time should be specified when creating standard libraries.

When creating standard libraries using the standard library configuration tool, the **cpu** option should be used to specify the same CPU type specified by the CPU/operating mode at compile time.

#### (2) Important information on options

The options relating to function interface listed below should always be the same at compile time and when building libraries. If object programs created using different options are linked, operation of the object program at runtime is not guaranteed.

- **cpu**
- **exception/noexception**
- **rtti = on/off**
- **regparam**
- **longreg/nolongreg**
- **structreg/nostructreg**
- **stack**
- **double=float**
- **byteenum**
- **pack**
- **bit\_order = left/right**
- **indirect = normal/extended** \*<sup>1</sup>

(It is possible to specify the indirect option to certain files of the whole source files, but a mixture of normal and extended is not allowed.)

- **ptr16**
- **sbr** \*<sup>2</sup>



- Notes: 1. indirect = extended is only available for the H8SX.
2. Only available for the H8SX.



# Section 10 C/C++ Language Specifications

## 10.1 Language Specifications

### 10.1.1 Compiler Specifications

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

#### (1) Environment

**Table 10.1 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 10.2 Identifier Specifications**

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

### (3) Characters

**Table 10.3 Character Specifications**

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

## (4) Integers

**Table 10.4 Integer Specifications**

No.	Item	Compiler Specifications
1	Representation and values of integers	See table 10.5.
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 value after conversion consists of the lower-order four bytes (if the post-conversion type is long), lower-order two bytes (if the post-conversion type is int/short), or lower-order byte (if the post-conversion type is char) of the integer value.
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 integral types with a negative value	Maintains sign bit.

**Table 10.5 Range of Integer Types and Values**

No.	Type	Value Range	Data Size
1	char	−128 to 127	1 byte
2	signed char	−128 to 127	1 byte
3	unsigned char	0 to 255	1 byte
4	short	−32768 to 32767	2 bytes
5	unsigned short	0 to 65535	2 bytes
6	int	−32768 to 32767	2 bytes
7	unsigned int	0 to 65535	2 bytes
8	long	−2147483648 to 2147483647	4 bytes
9	unsigned long	0 to 4294967295	4 bytes

## (5) Floating-point numbers

**Table 10.6 Floating-Point Number Specifications**

No.	Item	Compiler Specifications
1	Representation and values of floating-point type	There are three types of floating-point numbers: float, double, and long double types. See section 10.1.3, Floating-Point Number Specifications, for the internal representation of floating-point types and specifications for their conversion and operation. Table 10.7 shows the limits of floating-point type values that can be expressed.
2	Method of truncation when integers are converted into floating-point numbers that cannot accurately represent the actual value	
3	Methods of truncation or rounding when floating-point numbers are converted into shorter floating-point numbers	

**Table 10.7 Limits of Floating-Point Type Values**

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

Note: 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.

## (6) Arrays and Pointers

**Table 10.8 Array and Pointer Specifications**

No.	Item	Compiler Specifications
1	Integer type (size_t) required to hold maximum array size	unsigned int type (H8/300) unsigned int type (normal mode, H8S/2000 advanced mode with ptr16 option, H8S/2600 advanced mode with ptr16 option, H8SX middle mode, H8SX advanced mode with ptr16 option, H8SX maximum mode with ptr16 option) unsigned long type (H8/300H advanced mode, H8S/2000 advanced mode without ptr16 option, H8S/2600 advanced mode without ptr16 option, H8SX advanced mode without ptr16 option, H8SX maximum mode without ptr16 option)
2	Conversion from pointer type to integer type (pointer type size >= integer type size)	Value of least significant bytes 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 bytes of integer type
5	Conversion from integer type to pointer type (integer type size < pointer type size)	Zero extension
6	Integer type (ptrdiff_t) required to hold difference between pointers to members in the same array	int type (H8/300) int type (normal mode, H8SX middle mode, H8S/2000 advanced mode with ptr16 option, H8S/2600 advanced mode with ptr16 option, H8SX advanced mode with ptr16 option, H8SX maximum mode with ptr16 option) long type (H8/300H advanced mode, H8S/2000 advanced mode without ptr16 option, H8S/2600 advanced mode without ptr16 option, H8SX advanced mode without ptr16 option, H8SX maximum mode without ptr16 option)

## (7) Registers

**Table 10.9 Register Specifications**

No.	Item	Compiler Specifications
1	Registers to which register variables <sup>*5</sup> can be assigned	<div>H8/300</div> <div>Optimization: (R3)<sup>*1</sup>, R4, R5, R6</div> <div>No optimization: (R3)<sup>*1</sup>, R4, R5</div> <div>Others</div> <div>Optimization: (ER3)<sup>*1</sup>, ER4, ER5, ER6</div> <div>No optimization: (ER3)<sup>*1</sup>, ER4, ER5, ER6<sup>*4</sup></div>
2	Types of register variables <sup>*5</sup> that can be assigned to registers	char, unsigned char, short, unsigned short, int, unsigned int, long <sup>*2</sup> , unsigned long <sup>*2</sup> , float <sup>*2</sup> , pointer, reference, pointer to data member, structure data of 4 bytes or less <sup>*3</sup>

- Notes: 1. If the **noregexpansion** option is specified, no register variable is assigned to the register in the parentheses, ( ).
2. If the H8/300-series CPU is selected as the CPU, variables these of types be assigned to the register.
3. If the H8/300-series CPU is selected as the CPU, structure data of 2 bytes or less can be assigned.
4. Only if the H8SX-series and H8S CPU is selected as the CPU, register variable(s) can be assigned to ER6 even without optimization.
5. Allocation of a variable to a register is not affected by the register storage-class specifier. If the **enable\_register** option is specified, however, variables for which the register-storage class has been declared will be preferentially assigned to registers.



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

**Table 10.10 Class, Structure, Union, and Enumeration Type, and Bit Field Specifications**

No.	Item	Compiler Specifications
1	Referencing members in union type accessed by members of another type	Can be referenced but value cannot be guaranteed.
2	Boundary alignment of class members	Class consisting of only char type members are aligned to a 1-byte boundary. Other class members are aligned to a 2-byte boundary. For details on assignment, see section 10.1.2 (2), Compound Type (C), Class Type (C++).
3	Sign of bit fields of simple int type	signed int type
4	Order of bit fields within int type size	Assigned from the most significant bit. <sup>*1</sup> <sup>*2</sup>
5	Method of assignment when the size of a bit field assigned after a bit field is assigned within the int type size exceeds the remaining size in the int type	Assigned to the next int type area. <sup>*1</sup>
6	Permissible type specifiers in bit fields	char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long type
7	Integer type representing enumeration type	int, unsigned char <sup>*3</sup> , char <sup>*3</sup> type

Note: 1. For details of assignment of bit fields, see section 10.1.2 (3), Bit Fields.  
2. Specifying the `bit_order=right` option assigns bit fields from the least significant bit.  
3. When **bytenum** option is specified, type is unsigned char or char according to the value.

## (9) Qualifiers

**Table 10.11 Qualifier Specifications**

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

## (10) Declarations

**Table 10.12 Declaration Specifications**

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

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

- i. `int i;` Here, **i** has the `int` type (basic type) and the number of types modifying the basic type is 0.
- ii. `char *f( );` Here, **f** has a function type returning a pointer type to a `char` type (basic type), and the number of types modifying the basic type is 2.

## (11) Statements

**Table 10.13 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 10.14 Preprocessor Specifications**

No.	Item	Compiler Specifications
1	Whether the value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.	Preprocessor statement character constants are the same as the execution environment character set.
2	Method of locating include files	Files enclosed in "<" and ">" are read from the directory specified in the include option. If this specification is not made, files are read from the directory specified in the environment variable CH38.
3	Support for include files enclosed in double quotation marks	Supported. Include files are read from the current directory. If not found in the current directory, the file is searched for as described in 2, above.
4	White-space characters in string literals after code is expanded when string literals of real value parameters in a #define statement are white-space characters	A string of white-space characters is expanded as one white-space character.
5	Operation of #pragma statements	See section 10.2.1, #pragma Extension Specifiers and Keywords.
6	__DATE__ and __TIME__ values	A value is specified based on the host computer's timer at the start of compiling.

### 10.1.2 Internal Data Representation

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

1. Size  
Shows the memory size necessary to store the data.
2. Boundary alignment  
Restricts the addresses to which data is allocated. There are two types of alignment; 1-byte alignment in which data can be allocated to any address, and 2-byte alignment in which data is allocated to an even byte address.
3. Data range  
Shows the range of data of scalar type (C) or basic type (C++).
4. 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 10.15 shows internal representation of scalar-type data in C and basic type data in C++.

Table 10.15 Internal Representation of Scalar-Type and Basic-Type Data

Data Type	Size (bytes)	Alignment (bytes)	Sign	Data Range	
				Minimum Value	Maximum Value
<b>char</b>	1	1	Used	$-2^7$ (-128)	$2^7 - 1$ (127)
<b>signed char</b>	1	1	Used	$-2^7$ (-128)	$2^7 - 1$ (127)
<b>unsigned char</b>	1	1	Unused	0	$2^8 - 1$ (255)
<b>short</b>	2	2	Used	$-2^{15}$ (-32768)	$2^{15} - 1$ (32767)
<b>unsigned short</b>	2	2	Unused	0	$2^{16} - 1$ (65535)
<b>int</b>	2	2	Used	$-2^{15}$ (-32768)	$2^{15} - 1$ (32767)
<b>unsigned int</b>	2	2	Unused	0	$2^{16} - 1$ (65535)
<b>long</b>	4	2	Used	$-2^{31}$ (-2147483648)	$2^{31} - 1$ (2147483647)
<b>unsigned long</b>	4	2	Unused	0	$2^{32} - 1$ (4294967295)
<b>enum (the value range is -128 to 127 and byteenum option is specified)</b>	1	1	Used	$-2^7$ (-128)	$2^7 - 1$ (127)
<b>enum (the value range is 0 to 255 and byteenum option is specified)</b>	1	1	Unused	0	$2^8 - 1$ (255)
<b>enum (other than above)</b>	2	2	Used	$-2^{15}$ (-32768)	$2^{15} - 1$ (32767)
<b>bool</b> * <sup>1</sup>	1	1	Used	$-2^7$ (-128)	$2^7 - 1$ (127)
<b>float</b>	4	2	Used	$-\infty$	$+\infty$
<b>double</b> * <sup>2</sup> , <b>long double</b>	8	2	Used	$-\infty$	$+\infty$
<b>Pointer</b> (H8SX normal mode, H8SX middle mode, H8S/2600 normal mode, H8S/2000 normal mode, H8/300H normal mode, and H8/300)	2	2	Unused	0	$2^{16} - 1$ (65535)
<b>Pointer</b> * <sup>3</sup> (H8/300H advanced mode)	4	2	Unused	0	$2^{24} - 1$ (16777215)

**Table 10.15 Internal Representation of Scalar-Type and Basic-Type Data (cont)**

Data Type	Size (bytes)	Alignment (bytes)	Sign	Data Range	
				Minimum Value	Maximum Value
<b>Pointer*<sup>4</sup></b> (H8SX advanced mode, H8SX maximum mode, H8S/2600 advanced mode, and H8S/2000 advanced mode)	4	2	Unused	0	$2^{32} - 1$ (4294967295)
<b>Reference*<sup>1</sup></b> (H8SX normal mode, H8SX middle mode, H8S/2600 normal mode, H8S/2000 normal mode, H8/300H normal mode, and H8/300)	2	2	Unused	0	$2^{16} - 1$ (65535)
<b>Reference*<sup>1*3</sup></b> (H8/300H advanced mode)	4	2	Unused	0	$2^{24} - 1$ (16777215)
<b>Reference*<sup>1*4</sup></b> (H8SX advanced mode, H8SX maximum mode, H8S/2600 advanced mode, and H8S/2000 advanced mode)	4	2	Unused	0	$2^{32} - 1$ (4294967295)
<b>Pointer to data member*<sup>1</sup></b> (H8SX normal mode, H8SX middle mode, H8S/2600 normal mode, H8S/2000 normal mode, H8/300H normal mode, and H8/300)	2	2	Unused	0	$2^{16} - 1$ (65535)
<b>Pointer to data member*<sup>1*3</sup></b> (H8/300H advanced mode)	4	2	Unused	0	$2^{24} - 1$ (16777215)
<b>Pointer to data member*<sup>1*4</sup></b> (H8SX advanced mode, H8SX maximum mode, H8S/2600 advanced mode and H8S/2000 advanced mode)	4	2	Unused	0	$2^{32} - 1$ (4294967295)

Table 10.15 Internal Representation of Scalar-Type and Basic-Type Data (cont)

Data Type	Size (bytes)	Alignment (bytes)	Sign	Data Range	
				Minimum Value	Maximum Value
Pointer to function member* <sup>1</sup> * <sup>6</sup> (H8SX normal mode, H8S/2600 normal mode, H8S/2000 normal mode, H8/300H normal mode, and H8/300)	6	2	N/A	N/A	N/A
Pointer to function member* <sup>1</sup> * <sup>6</sup> (H8SX middle mode)	8	2	N/A	N/A	N/A
Pointer to function member* <sup>1</sup> * <sup>5</sup> * <sup>6</sup> (H8SX advanced mode, H8SX maximum mode, H8S/2600 advanced mode, H8S/2000 advanced mode, H8/300H advanced mode)	10	2	N/A	N/A	N/A
Pointer to virtual function member* <sup>1</sup> * <sup>6</sup> (H8SX normal mode, H8S/2600 normal mode, H8S/2000 normal mode, H8/300H normal mode, and H8/300)	6	2	N/A	N/A	N/A
Pointer to virtual function member* <sup>1</sup> * <sup>6</sup> (H8SX middle mode)	8	2	N/A	N/A	N/A
Pointer to virtual function member* <sup>1</sup> * <sup>5</sup> * <sup>6</sup> (H8SX advanced mode, H8SX maximum mode, H8S/2600 advanced mode, H8S/2000 advanced mode, and H8/300H advanced mode)	10	2	N/A	N/A	N/A

- Notes:
1. These data types are valid only with C++ compilation.
  2. The size of double type is 4 bytes if **double=float** is specified.
  3. The lower three bytes indicate address data and the highest byte has an indefinite value.
  4. In the H8SX advanced/maximum mode with **ptr16** option or **\_\_ptr16** keyword, the size is 2.
  5. In other H8/300H advanced mode with **ptr16** option, the size is 8.

6. Pointers to function and virtual function members are represented by classes in the following.

```
class _PMF{
    public:
        size_t delta;           //Object offset value.
        short index;           //Index in the virtual
                                //function table when
                                //the target function is a
                                //virtual function.

        union{
            int (*_deffun)();   //Address of a function when
                                //the target function is a
                                //non-virtual function.

            size_t vt_offset;   //Object offset value of the
                                //virtual function table
        };
};                               //when the target function
                                //is a virtual function.
```



## (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++.

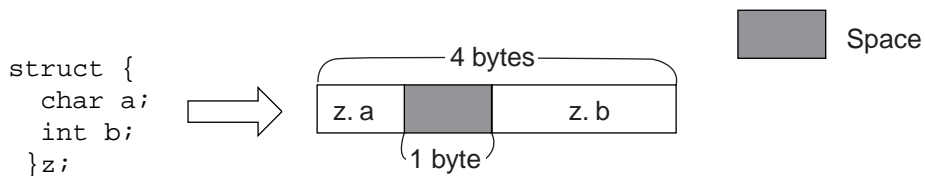
Table 10.16 shows internal representation of compound type and class type data.

**Table 10.16 Internal Representation of Compound Type and Class Type Data**

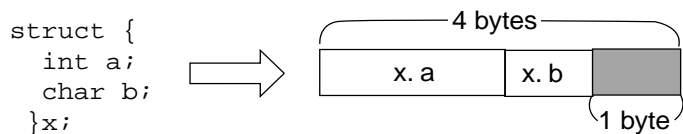
Data Type	Alignment (bytes)	Size (bytes)	Data Allocation Example
Array type	Array element alignment	Number of array elements × element size	<code>char a[10];</code> Alignment: 1 byte Size: 10 bytes
Structure type	Maximum structure member alignment	Total size of members. Refer to Structure Data Allocation, below.	<code>struct { char a,b; };</code> Alignment: 1 byte Size: 2 bytes
Union type	Maximum union member alignment	Maximum size of member. Refer to Union Data Allocation, below.	<code>union { char a,b; };</code> Alignment: 1 byte Size: 1 byte
Class type	1. Always 2 if a virtual function is included  2. Other than 1 above: maximum member alignment	Sum of data members, pointer to the virtual function table, and pointer to the virtual base class Refer to Class Data Allocation, below.	H8S/2600 advanced mode: <code>class B:public A { virtual void f(); };</code> Alignment: 2-byte Size: 6 bytes  <code>class A { char a; };</code> Alignment: 1-byte Size: 1 byte

## Structure Data Allocation:

- When structure members are allocated, 1-byte unused area may be generated between structure members to align them to their own boundaries.

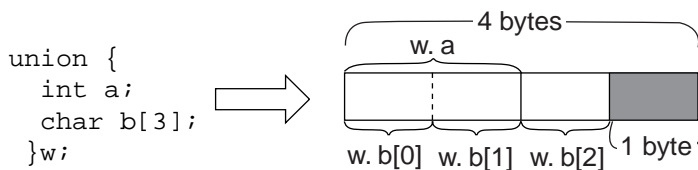


- If a structure has 2-byte alignment and the last member ends at an odd-byte address, the following one byte is included in this structure.



## Union Data Allocation:

- When a union has 2-byte alignment and its maximum member size is odd, the following one byte is included in this union.



## Class Data Allocation:

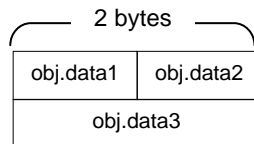
- For classes having no base class or virtual functions, data members are allocated according to the allocation rules of structure data.

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



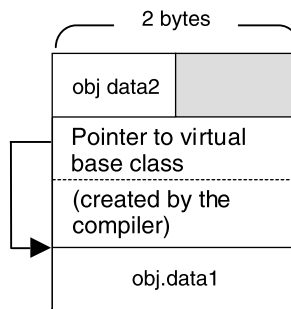
- If the start member for a class is 1-byte data and if the boundary alignment of the base class is 1, data members are allocated in order not to make a space.

```
class A{  
    char data1;  
};  
class B:public A{  
    char data2;  
    short data3;  
}obj;
```



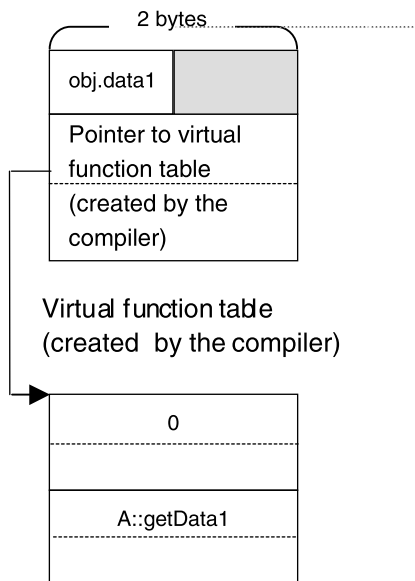
- For a class having a virtual base class, a pointer to the virtual base class is allocated.

```
class A{
    short data1;
};
class B: virtual protected A{
    char data2;
}obj;
```



- For a class having virtual functions, the compiler creates a virtual function table and allocates a pointer to the virtual function table.

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



- An example is shown for class having virtual base class, base class, and virtual functions.

```

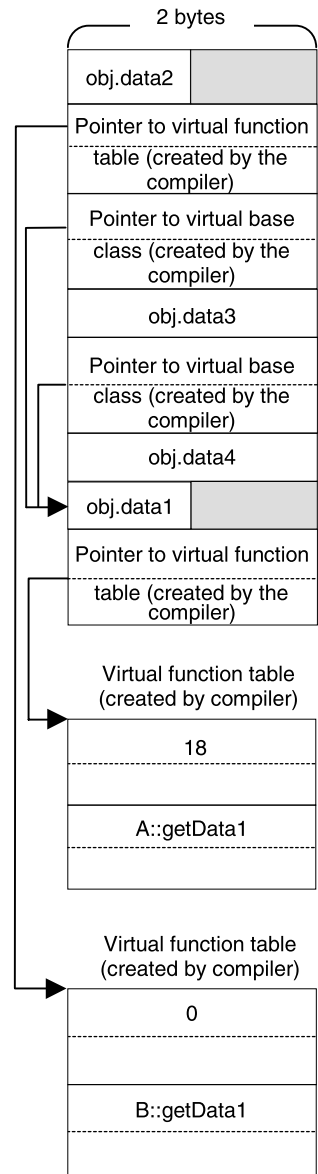
class A{
    char data1;
    virtual short getData1();
};

class B:virtual public A{
    char data2;
    char getData2();
    short getData1();
};

class C:virtual protected A{
    int data3;
};

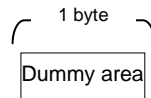
class D:virtual public A,public B,public C{
public:
    int data4;
    short getData1();
}obj;

```



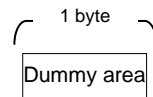
- For an empty class, a 1-byte dummy area is assigned.

```
class A{
    void fun();
}obj;
```



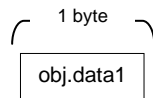
- For an empty class having an empty class as its base class, the dummy area is 1 byte.

```
class A{
    void fun();
};
class B: A{
    void sub();
};
```



- When the class size is 0, a dummy area for an empty class is allocated. For a base class or derived class with data members, or for a class with virtual functions, no dummy area is allocated.

```
class A{
    void fun();
};
class B: A{
    char data1;
}obj;
```



### (3) Bit Fields

A bit field is a member allocated with a specified size in a structure, union, or class. This part explains how bit fields are allocated.

**Bit Field Members:** Table 10.17 shows the specifications of bit field members.

**Table 10.17 Bit Field Member Specifications**

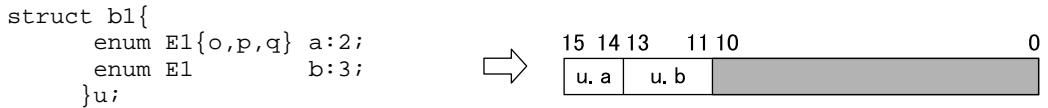
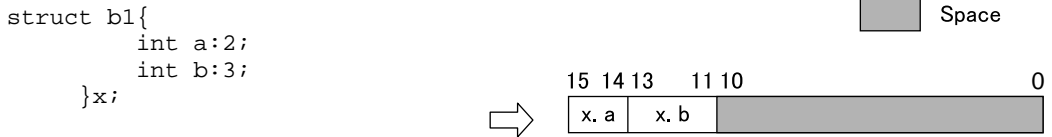
Item	Specifications
Type specifier allowed for bit fields	char, unsigned char short, unsigned short, int, unsigned int long, unsigned long
How to treat a sign when data is extended to the declared type* <sup>1</sup>	A bit field with no sign (unsigned is specified for type): Zero extension* <sup>2</sup>  A bit field with a sign (unsigned is not specified for type): Sign extension

Notes: 1. To use a member of a bit field, data in the bit field is extended to the declared type.  
2. Zero extension: Zeros are written to the upper bits to extend data.  
Sign extension: The most significant bit of a bit field is used as a sign and the sign is written to all higher-order bits to extend data.

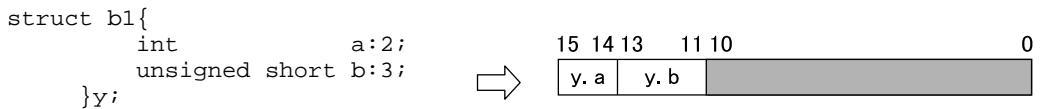
Note: One-bit bit field data with a sign (declared with signed) is interpreted as the sign, and can only represent 0 and -1. To represent 0 and 1, bit field data must be declared with unsigned.

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

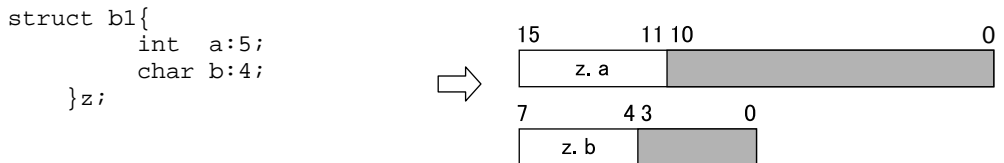
- Bit field members are placed in an area beginning from the left, that is, the most significant bit.



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



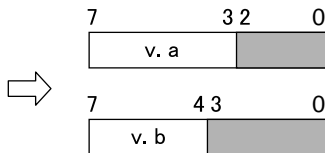
- Bit field members having type specifiers with different sizes are allocated to separate areas.





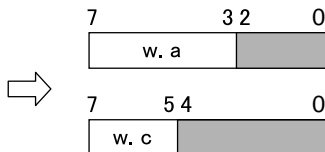
- If the number of remaining bits in an area is less than the next bit field size, though the type specifiers indicate the same size, the remaining area is not used and the next bit field is allocated to the next area.

```
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.

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



Note: When the H8SX is selected as the CPU, bit field members can be aligned to the lower-bit side. For details, refer to the description of the **bit\_order** option in section 2.2, Interpretation of Options, or the description of the **#pragma bit\_order** in section 10.2.1, #pragma Extension Specifiers and Keywords.

10.1.3 Floating-Point Number Specifications

(1) Internal Representation of Floating-Point Numbers

Floating-point numbers handled by this compiler are internally represented in the standard IEEE format. This section outlines the internal representation of floating-point numbers in the IEEE format.

(a) Format for internal representation

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

(b) Structure of internal representation

Figure 10.1 shows the structure of the internal representation of float, double, and long double types.

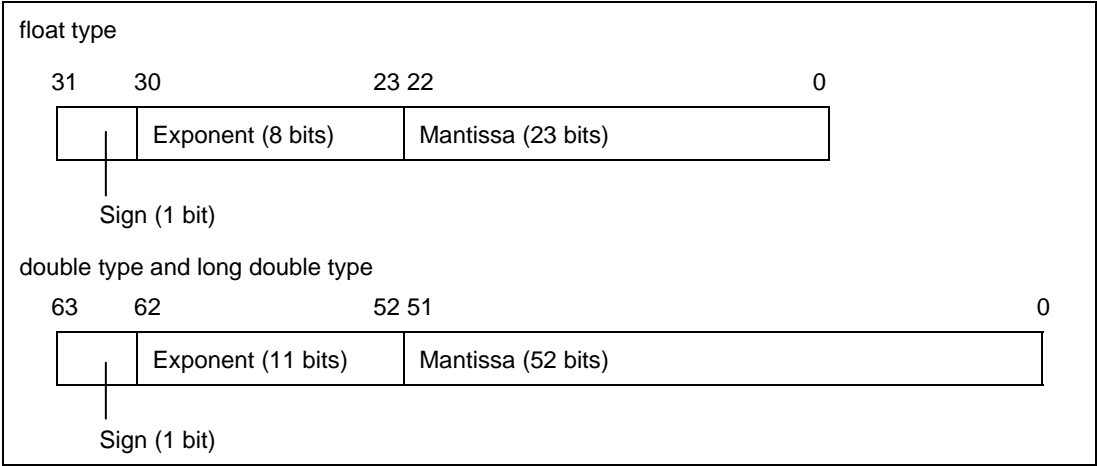


Figure 10.1 Structure of Internal Representation of Floating-Point Numbers

The internal representation format consists of the following parts:

- i. Sign  
Shows the sign of the floating-point number. 0 is positive, and 1 is negative.
- ii. Exponent  
Shows the exponent of the floating-point number to the power of 2.
- iii. Mantissa  
Shows the data corresponding to the significant digits of the floating-point number.

(c) Types of represented values of floating-point number

In addition to the normal real numbers, floating-point numbers can also represent values such as infinity. The following describes the types of values represented by floating-point numbers.

i. Normalized number

When the exponent is not 0 or not all bits are 1. Represents a normal real value.

ii. Denormalized number

When the exponent is 0 and the mantissa is other than 0. Represents a real value having a small absolute value.

iii. Zero

When the exponent and mantissa are 0. Represents the value 0.0.

iv. Infinity

When all bits of the exponent are 1 and the mantissa is 0. Represents infinity.

v. Not-a-number

When all bits of the exponents are 1 and the mantissa is other than 0. Represents the result of operation such as "0.0/0.0", " $\infty/\infty$ ", or " $\infty-\infty$ ", which does not correspond to a number or infinity.

Table 10.18 shows the types of values represented as floating-point numbers.

**Table 10.18 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 that can be 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.

## (2) float type

The float type is internally represented by a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa.

i. Normalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is between 1 and 254 ( $2^8-2$ ). The actual exponent is gained by subtracting 127 from this value. The range is between  $-126$  and  $127$ . The mantissa is between 0 and  $2^{23}-1$ . The actual mantissa is interpreted as the value of which the  $2^{23}$ 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:

[illegible]

Sign:  $-$

Exponent:  $10000000_{(2)} - 127 = 1$ , where  $_{(2)}$  indicates binary

Mantissa:  $1.11_{(2)} = 1.75$

Value:  $-1.75 \times 2^1 = -3.5$

- ii. Denormalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is 0 and the actual exponent is  $-126$ . The mantissa is between 1 and  $2^{23}-1$ , and the actual mantissa is interpreted as the value of which the  $2^{23}$ rd 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:

[illegible]

Sign: +

Exponent:  $-126$ 

**Mantissa:**  $0.11_{(2)} = 0.75$ , where  $_{(2)}$  indicates binary

Value:  $0.75 \times 2^{-126}$

iii. *Zero*

The sign is 0 (positive) or 1 (negative), indicating  $+0.0$  or  $-0.0$ , respectively. The exponent and mantissa are both 0.

+0.0 and -0.0 are both the value 0.0. See section 10.1.3 (4), Floating-Point Operation Specifications, for the functional differences deriving from the sign used with zero.

## iv. Infinity

The sign is 0 (positive) or 1 (negative), indicating  $+\infty$  or  $-\infty$ , respectively.

The exponent is 255 ( $2^8-1$ ).

The mantissa is 0.

v. Not-a-number

The exponent is 255 ( $2^8-1$ ).

The mantissa is a value other than 0.

Note: There are no stipulations regarding the mantissa values (other than 0) or the sign of not-a-number.

### (3) double type and long double type

The double type and the long double types are internally represented by a 1-bit sign, a 11-bit exponent, and a 52-bit mantissa.

- i. Normalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is between 1 and 2046 ( $2^{11}-2$ ). The actual exponent is gained by subtracting 1023 from this value. The range is between  $-1022$  and  $1023$ . The mantissa is between 0 and  $2^{52}-1$ . The actual mantissa is interpreted as the value of which the  $2^{52}$ nd bit is 1 and this bit is followed by the decimal point. Values of normalized numbers are as follows:

$$(-1)^{\text{sign}} \times 2^{\text{exponent}-1023} \times (1+(\text{mantissa}) \times 2^{-52})$$

Example:

[illegible]

Sign: +

Exponent:  $111111111_{(2)} - 1023 = 0$ , where  $_{(2)}$  indicates binary

**Mantissa:**  $1.111_{(2)} = 1.875$

Value:  $1.875 \times 2^0 = 1.875$



#### (4) Floating-Point Operation Specifications

This section describes the specifications for arithmetic operations on floating-point numbers in C/C++, and for conversion between the decimal representation of floating-point numbers and their internal representation during compilation and in C library processing.

##### (a) Specifications for arithmetic operations

###### i. Rounding of results

When the result of arithmetic operations on floating-point numbers exceeds the number of valid limit in the mantissa in internal representation, the result is rounded according to the following rules:

- a. The result is rounded toward the closer of the two internal representations of the approximating floating-point numbers.
- b. When the result is exactly between the two approximating floating-point numbers, it is rounded to the floating-point number of which the last digit of the mantissa is 0.

###### ii. Processing of overflows, underflows, and illegal operations

The following is performed in the event of an overflow, underflow, or illegal operation.

- a. In the case of an overflow, the result is a positive or negative infinity, depending on the sign of the result.
- b. In the case of an underflow, the result is a positive or negative zero, depending on the sign of the result.
- c. In the case of an illegal operation, in which infinity values of the opposite sign have been added, in which an infinity has been subtracted from another infinity of the same sign, in which zero has been multiplied by infinity, in which zero is divided by zero, or in which infinity is divided by infinity, the result is a not-a-number.
- d. For the cases above, error numbers are set to variable **errno** which indicates an error. For details on error numbers, refer to section 12.3, C Library Error Messages. Whether an error has occurred can be checked by the **errno** value.

**Note:** Operations are performed on constant expressions during compilation. If an overflow, underflow, or illegal operation occurs, a warning level error message is output.

iii. Notes on operations on special values

The following are notes on operations on special values (zero, infinity, and not-a-number).

- a. The sum of a positive zero and a negative zero is a positive zero.
- b. The difference between two zeros of the same sign is a positive zero.
- c. The result of operations that include not-a-number in one or both operands is always a not-a-number.
- d. In comparative operations, positive zeros and negative zeros are processed as equal.
- e. The result of comparative operations or equivalence operations where either one or both operands are not-a-number is true for "!=" and false in all other cases.

(b) Conversion between decimal and internal representation

This section describes the specifications for conversions between floating-point numbers in a source program and internal representation, and conversion by library functions between the decimal representation of floating-point numbers in ASCII strings and their internal representation.

- i. When converting from decimal to internal representation, the decimal value is first converted to its normalized form. The normalized form of a decimal value is " $\pm M \times 10^{\pm N}$ ", where M and N are in the following range:

- a. Normalized form of float type

$$0 \leq M \leq 10^9 - 1$$

$$0 \leq N \leq 99$$

- b. Normalized form of double and long double types

$$0 \leq M \leq 10^{17} - 1$$

$$0 \leq N \leq 999$$

If a decimal value cannot be converted to its normalized form, an overflow or underflow occurs. If the decimal representation contains more valid numerals than the normalized form, the trailing digits are truncated. In this case, a warning level error message is output at compilation and the corresponding error number is set in **errno** when the program is executed. For conversion to its normalized form, the original decimal representation must, in the form of an ASCII string, be within 511 characters. If not, an error occurs at compilation and the corresponding error number is set in **errno** when the program is executed. When converting from internal representation to decimal, the value is first converted to the normalized decimal form, then converted to an ASCII string according to the specified format.



ii. Conversion between normalized form of decimals and internal representation

When converting from the normalized form of decimals to internal representation, and vice versa, errors cannot be avoided when the exponent is large or small. The following describes the range within which conversion is accurate, and the error limits when the values are outside that range.

a. Range for accurate conversion

The rounding shown in (a) i, "Rounding of results" is correctly applied for floating-point numbers within the ranges shown below. No overflow or underflow will occur within these ranges.

(1) float types:  $0 \leq M \leq 10^9-1$ ,  $0 \leq N \leq 13$

(2) double and long double types:  $0 \leq M \leq 10^{17}-1$ ,  $0 \leq N \leq 27$

b. Error limits

The difference between the error that occurs when converting values that do not fall in the ranges shown in a. above and the error that occurs when rounding is correctly performed does not exceed 0.47 times the smallest digit of the valid numerals. If the value exceeds the ranges shown in a. above, an overflow or underflow may occur during conversion. In this case, a warning level error message is output during compilation, and the corresponding error number is set in **errno** when the program is executed.

## 10.1.4 Operator Evaluation Order

If an expression includes multiple operators, the evaluation order of these operators is determined according to the precedence indicated as positive value and the associativity indicated by right or left.

Table 10.19 shows each operator precedence and associativity.

**Table 10.19 Operator Precedence and Associativity**

Precedence	Operators	Associativity	Applicable Expression
1	( ) [ ] -> . ++ -- (postfix)	Left	Postfix expression
2	! ~ ++ -- (prefix) + - * & sizeof	Right	Monomial expression
3	(Type name)	Right	Cast expression
4	* / %	Left	Multiplicative expression
5	+ - -	Left	Additive expression
6	<< >>	Left	Shift expression
7	< <= > >=	Left	Relational expression
8	== !=	Left	Equality expression
9	&	Left	Bitwise AND expression
10	^	Left	Bitwise XOR expression
11		Left	Bitwise OR expression
12	&&	Left	Logical AND operation
13		Left	Logical OR expression
14	?:	Left	Conditional expression
15	= += -= *= /= %= <<= >>= &=  = ^=	Right	Assignment expression
16	,	Left	Comma expression

## 10.2 Extended Functions

The compiler supports the following three kinds of extended specifications:

- #pragma extension and keywords
- Section address operator
- Intrinsic functions

### 10.2.1 #pragma Extension Specifiers and Keywords

Tables 10.20 to 10.22 list #**pragma** extension and keywords.

**Table 10.20 #pragma Extension Specifier Related to Memory Allocation**

#pragma Extension Specifier	Keyword	Function
#pragma stacksize	—	Creates a stack section
#pragma section, #pragma abs8 section, #pragma abs16 section, #pragma indirect section	—	Switches sections
#pragma abs8, #pragma abs16	_abs8, _abs16	Specifies a variable to access in short absolute addressing mode
—	_near8, _near16	Specifies an address calculation size for array and structure
—	_ptr16	Specifies the pointer size
#pragma bit_order	—	Specifies the order of bit field assignment

**Table 10.21 Extended Specifications Related to Functions**

<b>#pragma Extension Specifier</b>	<b>Keyword</b>	<b>Function</b>
#pragma interrupt	_ _interrupt	Creates an interrupt function
#pragma entry	_ _entry	Creates an entry function
#pragma indirect	_ _indirect	Specifies a function to be called in memory indirect addressing mode
–	_ <i>indirectex</i>	Specifies a function to be called in the extended memory indirect addressing mode
#pragma inline	_ _inline	Performs inline expansion of functions
#pragma inline_asm	–	Expands an assembly-language description function.
#pragma regsave, #pragma noregsave	_ <i>regsave</i> , _ <i>noregsave</i>	Controls generation of code to save and restore register contents.
–	_ <i>regparam2</i> , _ <i>regparam3</i>	Specifies the number of parameter registers.
#pragma option	–	Specifies an optimization option on function by function basis.

**Table 10.22 Other Extended Specifications**

<b>#pragma Extension Specifier</b>	<b>Keyword</b>	<b>Function</b>
#pragma asm, #pragma endasm	–	Embeds assembly-language instructions.
–	_ _asm	Performs assembly functions
#pragma global_register	_ <i>global</i> register	Allocates global variables to registers
#pragma pack 1, #pragma pack 2, #pragma unpack	–	Specifies the boundary alignment of structures, unions, and classes.
–	_ _evenaccess	Specifies an even byte access.
#pragma address	–	Allocates a variable to the specified address.

Note: The first keyword or **#pragma** extension specified for a function or variable is valid. Once an attribute has been specified, a different attribute cannot be specified for the same function or variable. It is also not possible to specify both a #pragma extension and keyword for the same variable.

Error examples:

```
// Different keywords cannot be specified for a prototype declaration and definition.
```

```
_ _regsave void func(void);
_ _interrupt void func(void) {}
```

```
// Different attributes cannot be specified in pragma in the same way.
```

```
#pragma regsave func
_ _interrupt void func(void) {}
```

To specify more than one attribute for one function or variable, specify all the attributes at the same time as a combination of keywords in a declaration or definition.

Example that will be compiled correctly:

```
// Keywords can be specified together at the same time in a declaration or definition.
```

```
_ _regsave _ _interrupt void func(void);
void func(void) {}
```

## #pragma stacksize

Description: Creates the stack section S whose size is <constant>.

Example:	#pragma stacksize 100	<Code expansion example>
		.SECTION S,STACK
		.RES.W 50

Remarks:

1. Must specify an even number for stack size <constant>
2. **#pragma stacksize** can only be specified once within a file

## #pragma section

## #pragma abs8 section

## #pragma abs16 section

## #pragma indirect section

Description Format: #pragma section [{<name> | <numeric value>}]  
 #pragma abs8 section [{<name> | <numeric value>}]  
 #pragma abs16 section [{<name> | <numeric value>}]  
 #pragma indirect section [{<name> | <numeric value>}]

Description: Switches the section to be output by the compiler.  
Table 10.23 lists the default section names and section names after switching sections.

**Table 10.23 Section Switching and Section Name**

Target Area		Specification	Default Section Name	After Switching Section
Program area		#pragma section <xx>	P*	P<xx>
Constant area			C*	C<xx>
Initialized data area			D*	D<xx>
Uninitialized data area			B*	B<xx>
8-bit absolute address area	Constant area	#pragma abs8 section <xx>	\$ABS8C	\$ABS8C<xx>
	Initialized data area		\$ABS8D	\$ABS8D<xx>
	Uninitialized data area		\$ABS8B	\$ABS8B<xx>
16-bit absolute address area	Constant area	#pragma abs16 section <xx>	\$ABS16C	\$ABS16C<xx>
	Initialized data area		\$ABS16D	\$ABS16D<xx>
	Uninitialized data area		\$ABS16B	\$ABS16B<xx>
Area in memory indirect addressing mode	Function address area	#pragma indirect section <xx>	\$INDIRECT \$EXINDIRECT	\$INDIRECT<xx> \$EXINDIRECT<xx>

Note: The default section name can be modified by the **section** option.

If <name> or <numeric value> is not specified, the default section names will be used.

Example:

```
#pragma section abc
int a;                /* a is assigned to section Babc */
const int c=1;        /* c is assigned to section Cabc */
void f(void)          /* f is assigned to section Pabc */
{
    a=c;
}
#pragma section
int b;                /* b is assigned to section B */
void g(void)          /* g is assigned to section P */
{
    b=c;
}
```

Remarks:

1. Declare **#pragma section**, **#pragma abs8 section**, **#pragma abs16 section**, and **#pragma indirect section** outside function definitions.
2. Up to 64 names can be declared for each section within a file.



```

#pragma abs8
#pragma abs16
_abs8
_abs16

```

Description Format: **#pragma abs8** (<variable name> [...])  
**#pragma abs16** (<variable name> [...])  
**\_abs8** <type specifier><variable name>  
<type specifier>**\_abs8**<variable name>  
**\_abs16** <type specifier><variable name>  
<type specifier>**\_abs16**<variable name>

Description: Declares variables to allocate in the 8-bit and 16-bit absolute address area.

1. The variables declared in **#pragma abs8** and **\_abs8** are output to sections “\$ABS8C”, “\$ABS8D”, and “\$ABS8B”, and the code to access them in 8-bit absolute addressing mode (@**aa:8**) is generated.
2. The variables declared in **#pragma abs16** and **\_abs16** are output to sections “\$ABS16C”, “\$ABS16D”, and “\$ABS16B”, and the code to access them in 16-bit absolute addressing mode (@**aa:16**) is generated.
3. For details on section name switching, refer to description on **#pragma abs8 section** and **#pragma abs16 section** above.

Example:

```

#pragma abs8(c1)
#pragma abs16(i1)
char c1;                /* c1 is assigned to $ABS8B          */
int i1;                 /* i1 is assigned to $ABS16B         */
char _abs8 c2;          /* c2 is assigned to $ABS8B          */
char _abs16 i2;         /* i2 is assigned to $ABS16B         */
long l;                 /* l is assigned to B                */
void f(void){
    c1=c2=10;           /* c1 and c2 are accessed by 8-bit   */
                        /* absolute address                  */
    i1=i2=100;          /* i1 and i2 are accessed by 16-bit  */
                        /* absolute address                  */
    l=1000;              /* l is accessed by 32-bit           */
                        /* absolute address                  */
}

```

Remarks:

1. The variables in the definition and declaration after the **#pragma abs8** or **#pragma abs16** declaration will be treated as the target variables.
2. Only variables to be allocated to the static area can be specified with **#pragma abs8**, **\_\_abs8**, **#pragma abs16**, and **\_\_abs16**.
3. Up to 63 variables can be specified in one **#pragma abs8** or **#pragma abs16** directive.
4. The variables specified with **#pragma abs8**, **\_\_abs8**, **#pragma abs16**, or **\_\_abs16** are output to section \$ABS8C, \$ABS8D, \$ABS8B, \$ABS16C, \$ABS16D, or \$ABS16B when neither **#pragma abs8** section <xx> nor **#pragma abs16** section <xx> is used. Allocate the target section to the 8-bit or 16-bit absolute addressing area at linkage.
5. If the variables declared by **#pragma abs8** cannot be accessed in 1 byte units, an error will occur. Declare a variable, array, or structure that is aligned to a 1-byte boundary.

## **\_\_near8** **\_\_near16**

Description Format: <type specifier> **\_\_near8** <variable name>  
 \_\_near8 <type specifier> <variable name>  
 <type specifier> **\_\_near16** <variable name>  
 \_\_near16 <type specifier> <variable name>

Description: Specifies an array or structure whose address can be calculated by an 8-bit or 16-bit address. When **\_\_near8** is specified, calculates an array or structure address using the lower 1 byte. When **\_\_near16** is specified, calculates an array or structure address using the lower 2 bytes.

Example:	<p>When <b>__near8</b> is not specified</p> <pre> struct a{     short a1;     short a2,a3; }; struct a aa[10]; void f(){     int i;     for(i=0;i&lt;11;i++)         aa[i].a1 = 0; } </pre> <p>&lt;Code expansion example&gt;</p> <pre> MOV.L    #_aa,ER1 SUB.L    ER0,ER0 </pre> <p>Ld:</p> <pre> MOV.W    R0,@ER1 INC.W    #H'1,E0 ADDS.L   #H'4,ER1 INC.L    #H'2,ER1 CMP.W    #H'B,E0 BLT      Ld:8 RTS </pre>	<p>When <b>__near8</b> is specified</p> <pre> struct a{     short a1;     short a2,a3; }; struct a __near8 aa[10]; void f(){     int i;     for(i=0;i&lt;11;i++)         aa[i].a1 =0; } </pre> <p>&lt;Code expansion example&gt;</p> <pre> MOV.L    #_aa,ER1 SUB.L    ER0,ER0 </pre> <p>Ld:</p> <pre> MOV.W    R0,@ER1 INC.W    #H'1,E0 ADD.B    #H'6,R1L CMP.W    #H'B,E0 BLT      Ld:8 RTS </pre>
----------	--	---

Remarks:

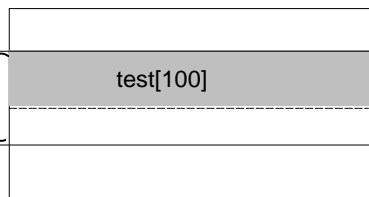
1. When `__near8` or `__near16` is specified for an array or a structure, that array or structure must be allocated to the area where no overflow occurs during 8-bit or 16-bit address calculation.
2. If an array or a structure to which `__near8` or `__near16` is specified is not allocated correctly, an error occurs at linkage.
3. If a variable is not allocated on the 8-bit or 16-bit address boundary, the compiler operation cannot be guaranteed. In this case, `__near8` or `__near16` cannot be specified.

```
struct b{  
    char buffer1;  
    char buffer2;  
};  
struct b __near8 test[100];
```

Allocate so that  
the address  
can be calculated  
within 1 byte.

H'400

H'500



## **\_\_ptr16**

Description Format: <type specifier> \_\_ptr16 <\*>

Description: Specifies the pointer size as two bytes. A pointer value will be specified by two signed bytes, and the target to be accessed must be allocated to the 16-bit absolute address area.

Example:	<b>When __ptr16 is not specified</b>	<b>When __ptr16 is specified</b>
	<pre>abs16 int a; int *b;</pre>	<pre>__abs16 int a; int __ptr16 *b;</pre>
	<pre>func() {     b = &amp;a; }</pre>	<pre>func() {     b = (int __ptr16 *)&amp;a; }</pre>
	<b>&lt;Code expansion example&gt;</b>	<b>&lt;Code expansion example&gt;</b>
	<pre>_func:     mov.l    #_a,er0     mov.l    er0,@_b:32</pre>	<pre>_func:     mov.l    #_a,er0     mov.w    r0,@_b:16</pre>

Remarks:

1. This keyword must be specified before a unary operator \*.
2. This keyword is effective only with H8SX advanced mode, H8SX maximum mode, H8S/2600 advanced mode, or H8S/2000 advanced mode.

## #pragma bit\_order

Description Format: #pragma bit\_order [{left|right}]

Description: Switches the order of bit field assignment.

When **left** is specified, bit field members are assigned from the most significant bit side. When **right** is specified, members are assigned from the least significant bit side.

The default setting is the interpretation of the **bit\_order** option.

If #pragma bit\_order is specified without left or right specifier, the interpretation of the **bit\_order** option is effective below the line.

Example:

```
#pragma bit_order left
typedef struct{
    unsigned char a:2;
    unsigned char b:3;
}x;
```




Diagram illustrating bit field assignment for #pragma bit\_order left. The bit field is divided into two parts: x.a (bits 7-6) and x.b (bits 5-3). Bit 2 is marked as Space.

```
#pragma bit_order right
typedef struct{
    unsigned char a:2;
    unsigned char b:3;
}y;
```

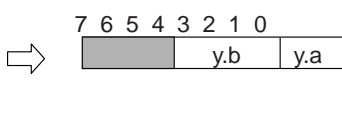


Diagram illustrating bit field assignment for #pragma bit\_order right. The bit field is divided into two parts: y.b (bits 5-3) and y.a (bits 2-0). Bit 7 is marked as Space.

```
// Member with different size
#pragma bit_order right
typedef struct{
    unsigned int a:3;
    unsigned char b:4;
}z;
```

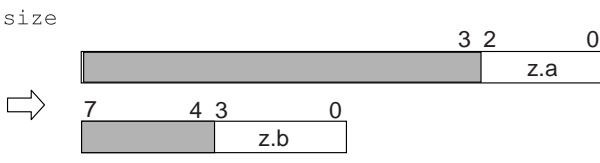


Diagram illustrating bit field assignment for #pragma bit\_order right with different sizes. The bit field is divided into two parts: z.a (bits 3-0) and z.b (bits 7-4). Bit 2 is marked as Space.

```
// Size of a type is exceeded
#pragma bit_order right
typedef struct{
    unsigned char a:5;
    unsigned char b:4;
}v;
```

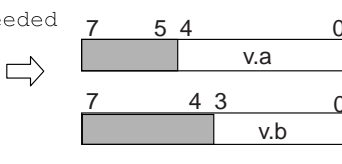


Diagram illustrating bit field assignment for #pragma bit\_order right when size is exceeded. The bit field is divided into two parts: v.a (bits 5-0) and v.b (bits 7-4). Bit 2 is marked as Space.

- Remarks:
1. The specified order of assignment is valid until it is switched again.
  2. The order of bit field assignment can also be specified by a compiler option. For details, refer to section 2.2.2, Object Options.
  3. For details of bit field, refer to section 10.1.2 (3), Bit Fields.

## (2) Extended Specifications Related to Functions

### #pragma interrupt

#### \_\_interrupt

Description Format: #pragma interrupt ( <function name>[(interrupt specification)][,...] )  
\_\_interrupt[(interrupt specification)]<type specifier><function name>  
<type specifier> \_\_interrupt[(interrupt specification)]<function name>

Description: Declares an interrupt function.  
Table 10.24 lists interrupt specifications.

**Table 10.24 Interrupt Specifications**

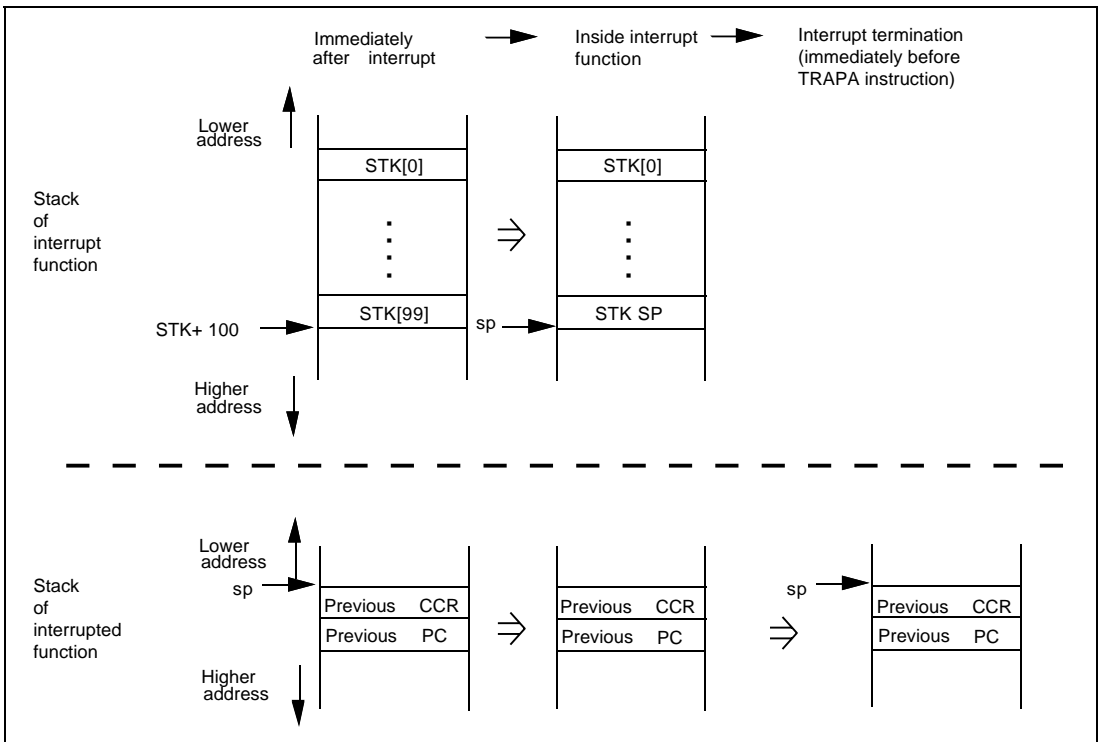
Item	Form	Options	Specifications
Stack switching	sp=	{<variable>  &<variable>  <constant>  <variable>+<constant>  &<variable>+<constant> }	The address of a new stack is specified with a variable or a constant. <variable>: Variable (pointer type) &<variable>: Variable (object type) address <constant>: Constant value
Trap-instruction return	tn=	<constant>	Termination is specified by the TRAPA instruction <constant>: Constant value (trap vector number)
Interrupt function termination	sy=	{<function name>  <constant>  \$<function name> }	Termination is specified by a jump instruction to an interrupt function <function name>: Interrupt function name <constant>: Absolute address \$<function name>: Interrupt function name without an underscore ( _ )
Vector table specification	vect=	<vector number>	A vector number to which an interrupt function address is assigned is specified

1. An interrupt function declared by **#pragma interrupt** preserves the register values of R0,R1and (R2 with regparam=3) in H8/300 or ER0 ER1 and (ER3 with regparam=3) in the other CPU before processing and executes the RTE instruction at the end of the function.
2. If the trap-instruction return (**tn=**) is specified, the TRAPA instruction is executed at the end of the function.

Example:

```
extern char STK[100];
#pragma interrupt ( f(sp=STK+100, tn=2) )
                    (1)      (2)
__interrupt(sp=STK+100, tn=2) void g(void);
                    (1)      (2)
```

1. STK+100 is set as the stack pointer used by interrupt functions **f** and **g**.
2. After the interrupt function has completed its processing, trap exception processing starts by TRAPA #2. The SP at the beginning of the trap exception processing is shown in the figure below. In the trap routine, the previous PC, CCR (condition code register), and EXR (extended control register: only for the H8SX, H8S/2600 and H8S/2000) must be popped from the stack by the RTE instruction, then control must be returned from the interrupt function.



**Figure 10.2 Stack Processing by an Interrupt Function**

3. When an interrupt function termination is specified (**sy=**), the program jumps to the address specified by the **JMP** instruction. For the function name of the interrupt function termination specification, **\$ + <function name>** can be specified as well as **<function name>**. If **\$ + <function name>** is specified, no underscore character (**\_**) to mean an external identifier is added at the beginning of the function name.



Example:

```
#pragma interrupt (f1(sy=$f2)) /* No underscore (_) */
                                /* added to the head */
                                /* of function name */
void f2(void) /* Returns by JMP @f2:24*/
{
    :
}
```

4. When a vector table is specified (**vect=**), a function address is assigned to the address corresponding to the vector number.

Example: (cpu=300)

```
#pragma interrupt (f2(vect=4)) /* Function f2 address */
                                /* is assigned to */
void f1(void) /* address 8 */
              /* (vector number 4) */
{
    :
}
```

5. An interrupt function with no interrupt specification is processed as a simple interrupt function.

Remarks:

1. The functions in the definition and declaration after the **#pragma interrupt** declaration will be treated as the interrupt functions.

Example:

```
#pragma interrupt (A::f) /* The functions in the          */
                        /* definition and declaration */
                        /* after #pragma interrupt      */
                        /* declaration will be         */
                        /* treated as the               */
                        /* interrupt functions          */

class A{
public:
    static void f(void); /* Static member function */
                        /* handled as interrupt    */
                        /* function                */

};
void A::f(void)
{
    ...
}
```

2. Functions that can be defined as an interrupt function are global functions and static member functions. The function must return only void data. The return statement cannot have a return value. If attempted, an error is output.

Example:

```
#pragma interrupt(f1(sp=100),f2)
void f1(void) /* Correct declaration. */
{
    ...
}
int f2(void) /* When the return type is not void, */
            /* a error is output. */
{
    ...
}
```

3. A function declared as an interrupt function cannot be called within the program. If attempted, an error is output. However, if the function is called within a program which does not declare it to be an interrupt function, an error is not output but correct program execution is not guaranteed.

Example:

```
#pragma interrupt(f1)
void f1(void)
{
    ...
}
int f2(void)/* Function f1 is declared as interrupt */
           /* function and an error is output.      */
{
    f1();
}
```

4. A program can refer to a function declared as an interrupt function if the function is not explicitly called.

Example:

```
#pragma interrupt f
void f(void)
{
    ...
}
void (*VTBL)(void)={f}; /* Correct compilation is */
                        /* guaranteed for references */
                        /* except for function calls */
```

5. Up to 63 functions can be declared in one **#pragma interrupt** directive line. Stack switching specification and trap-instruction return specifications, and stack switching specification and interrupt function termination specifications can be specified at the same time.

If stack switching is specified for the interrupt function, the size of the area to save the contents of the previous SP and ER0 (R0 for H8/300) which is used to calculate the new SP value is included in the Linkage Area Size in the symbol allocation information shown in the compile listing.

## #pragma entry

Description Format:

```
#pragma entry <function name>[<entry specification>]
__entry [(<entry specification>)] <type specifier> <function name>
<type specifier> __entry[<entry specification>] <function name>
<entry specification>: {sp=<constant> | vect=<vector number>}
```

**Description:** Handles the function specified in <function name> as the entry function.

1. Outputs the code for initial setting of the stack pointer at the beginning of the entry function when **sp** is specified. The `<constant>` specified by the **sp** is used as the stack-pointer initial value.

Example: (cpu=300)                      <Code expansion example>

```
#pragma entry INIT(sp=0x8000)      .SECTION P,CODE  
void INIT()                        INIT:  
{                                  MOV.W      #H'8000,SP  
    :                               :  
}  

```

2. If no **sp** is specified, the end address of the stack section created by the **#pragmastacksize** is used as the stack-pointer initial value.

Example: (cpu=300)                      <Code expansion example>

```
#pragma stacksize 100      .SECTION S,STACK
#pragma entry INIT         .RES.W   50
void INIT()               .SECTION P,CODE
{                          _INIT:
    :                     MOV.W #STARTOF S + SIZEOF S,SP
}                          :
```

3. If no **sp** is specified and no **#pragma stacksize** is declared in the program, section S with size 0 is created, and the end address of the S section is used as the stack-pointer initial value. Declare **#pragma stacksize** in the program or use the start option to allocate section S to the correct address at linkage.

Example: (cpu=300)

<Code expansion example>

```
#pragma entry INIT      .SECTION S,STACK
                        ; Creates section S
                        ; with size 0.
void INIT()             .SECTION P,CODE
{                       _INIT:
    :                   MOV.W    #STARTOF S + SIZEOF S,SP
}
```

4. When **vect** is specified, a function address is assigned to the address corresponding to the vector number.

Example: (cpu=300)

```
#pragma entry INIT(vect=0) /* Function INIT address */
                          /* assigned to address 0 */
void INIT()
{
    :
}
```

<Code expansion example>

```
.SECTION VECT0, DATA, LOCATE=0
.DATA.W _INIT
```

5. Does not output the save and restore code of the registers at the entry and exit of the entry function.

6. When the CPU type is H8SX and an option or environment variable has been used to change the SBR value, a function for which **#pragma entry** has been specified will include automatic setting of the SBR value.

Example: (cpu=H8SXA)

```
// -SBR=0xFF00 is specified for compilation as an example
#pragma entry INIT
void INIT()

:
}
```

<Code expansion example>

```
.SECTION P, CODE
__INIT:
MOV.L          #H'FF00, ER3
LDC.L          ER3, SBR
:
```

Remarks:

1. Specify the **#pragma entry** <function name> before declaring the <function name>.
2. Keywords can be specified for both declaration and definition. Note, however, that **SP** or **vect** cannot be specified with a keyword specified for a function declaration.
3. Only one entry function can be specified within one load module.

## **#pragma indirect** **\_\_indirect**

Description Format: #pragma indirect (<function name>[(vect=<vector number>)][,...])  
<type specifier> \_\_indirect[(vect=<vector number>)] <function name>  
\_\_indirect[(vect=<vector number>)] <type specifier> <function name>

Description: Specifies the functions to be called in memory indirect addressing mode (@@aa:8).

1. The function declared by the **#pragma indirect** or **\_\_indirect** statement is called in the format of **JSR @@\$function\_name:8**.  
When vect is specified, the function address is assigned to the address corresponding to the vector number.  
When vect is not specified for the function declared in memory indirect function call statement, the “\$function\_name” label and the function address are stored in the section “\$INDIRECT” as the address table for memory indirect function calls.
2. For details on section switching, refer to the description of **#pragma indirect section** in section 10.2.1 (1), Extended Specifications Related to Memory Allocation.

Example: (cpu=300)

```
_ _indirect(vect=5) char f(void); /* Function f address is */
                                   /* assigned to address 10 */

char f(void)
{
    ...
}
#pragma indirect (g)
unsigned char g(void) /* $g is created in section $INDIRECT */
                    /* and stores the function g address */

{
    ...
}
void sub()
{
    f();          /* Function is called in @@$f:8 memory */
                 /* indirect addressing mode */
    g();          /* Function is called in @@$g:8 memory */
                 /* indirect addressing mode */
}
```

- Remarks:
1. The functions in the first definition and declaration after the **#pragma indirect** declaration having the same function names as in the **#pragma indirect** declaration are treated as the target functions.
  2. Up to 63 functions can be specified in one **#pragma indirect** directive.
  3. Up to 128 functions can be specified in the normal and H8/300 mode and up to 64 in the other modes in total. The address table section that has been created without **vect** specification must be allocated within the range from H'0x0000 to 0x00FF at linkage.
  4. Run-time routines can be called in the memory indirect addressing mode by declaration of **#include <indirect.h>**. To select a run-time routine to be called in the memory indirect addressing mode, change unnecessary **#pragma indirect** statements into comments inside **indirect.h**.



## **`__indirect_ex`**

Description Format: <type specifier> `__indirect_ex`[(vect=<vector number>)] <function name>  
`__indirect_ex`[(vect=<vector number>)] <type specifier> <function name>

Description: Declares a function to be called in the extended memory indirect addressing mode (@@vec).

The function declared by the `__indirect_ex` statement is called in the format of **JSR @@ \$\$function\_name:7**.

When **vect** is specified, the function address is assigned to the address corresponding to the vector number. The vector number is 128 to 255.

When **vect** is not specified for the function declared in an extended memory indirect function call statement, the “\$\$function\_name” label and the function address are stored in the section “\$EXINDIRECT” as the address table for extended memory indirect function calls.

Example: (cpu=300)

```
__indirect_ex(vect=128)char f1(void);/*Function f1 address is*/  
                                /*assigned to address 0x200 */  
  
char f1(void)  
{  
    ...  
}  
  
void sub1(void)  
{  
    f1( );    /* Function is called in @$$f1:7,          */  
              /* extended memory indirect addressing mode */  
}
```

Remarks:

1. This keyword is valid only when the CPU is H8SX.
2. Up to 128 functions can be specified to `__indirect_ex` in the whole program. The address table section (\$EXINDRECT) that has been created without the **vect** specification must be allocated within the range from 0x0100 to 0x01FF for H8SX normal mode, or from 0x000200 to 0x0003FF for H8SX middle mode, H8SX advanced mode and H8SX maximum mode.

## **#pragma inline** **\_\_inline**

Description Format: **#pragma inline** (<function name>[,...])  
                  **\_\_inline** <type specifier> <function name>  
                  <type specifier> **\_\_inline** <function name>

Description: Declares functions for which inline expansion is performed.

When **#pragma inline** declares a function, the function code is directly generated at the place where it is called. The code for calling the function by the JSR or BSR instruction is not generated.

Example:

```
#pragma inline (f)      /* Declares function f as */
                        /* an inline function      */

int a,b,c;
int f(int x,int y)
{
    return x+y;
}

void sub(void)
{
    a=f(b,c);           /* Expanded directly to      */
                        /* code a=b+c                */
}
```

- Remarks:
1. The functions in the first definition and declaration after the **#pragma inline** declaration having the same function names as in the **#pragma inline** declaration are treated as the target functions.
  2. Up to 63 functions can be specified in one **#pragma inline** directive.
  3. When the function declared by **#pragma inline** or **\_\_inline** satisfies one of the following conditions, inline expansion will not be performed:  
The function is defined before the **#pragma inline** or **\_\_inline** specification.  
A variable number of arguments is used.  
A parameter address is referenced.  
The actual parameter type does not match the formal parameter type.  
The maximum size of inline expansion is exceeded.  
An address of a function to be expanded is used to call the function.
  4. When a source program file includes an inline function description, be sure to specify **static** before the function declaration because an external definition is generated even for a function specified by **#pragma inline** or **\_\_inline**. If **static** or **inline(C++)** is specified, an external definition will not be created.

## #pragma inline\_asm

Description Format: #pragma inline\_asm (<function name>[,...])

<function name>: Do not specify a C++ member function or an overloaded function.

Description: Performs inline expansion for the functions written in assembly language declared by **#pragma inline\_asm**.

Parameters of a function that is written in assembly language are referenced in an **inline\_asm** function because they are stacked or stored in registers in the same way as general function calls. The return value of an inline function written in assembly language should be set to (E)R0.

Example:

```
#pragma inline_asm(shlu)
extern unsigned int x;
static unsigned int shlu(unsigned int a)
{
    /* Function shlu is deleted */
    SHLL.W   R0
    BCC      ?L1
    SUB.W    R0,R0
    ?L1:     /* Local label starts with ? */
}
void main(void)
{
    x = shlu(x) /* Inline expansion is performed */
               /* within the main function */
}
```

Remarks:

1. Compile the program using the object-type specification option **code=asmcode**.
2. The functions in the definitions after the **#pragma inline\_asm** will be treated as the target functions.
3. Specify **#pragma inline\_asm** before the definition of the function. External definition will be generated for functions specified by **#pragma inline\_asm**. When a source program includes the same inline function description, be sure to specify **static** before the function declaration. If **static** is specified, an external definition will not be created.

4. Use local labels in an intrinsic inline function written in assembly language. For details on local labels, refer to section 11, Assembly Specifications.
5. When using registers ER2 to ER6 in an intrinsic inline function written in assembly language, the contents of these registers must be saved and restored at the beginning and end of the function.
6. Do not use **RTS** at the end of an inline function written in the assembly language.
7. When the compiler outputs an assembly program, and inline expansion is performed to the program, the assembler may display error message “402 ILLEGAL VALUE IN OPERAND”. This is the compiler generates the code without displacement. So be assembled it with optimize option. Or use the **JMP** instruction and modify the assembly-language program to satisfy the required branch width if necessary.

Example: Before modification	After modification
:	:
BEQ L1	BNE Ld
:	JMP L1
	Ld:

**#pragma regsave**  
**#pragma noregsave**  
**\_\_regsave**  
**\_\_noregsave**

Description Format: **#pragma regsave** (<function name>[,...])  
**#pragma noregsave** (<function name>[,...])  
**\_\_regsave** <function specifier> <function name>  
<function specifier> **\_\_regsave** <function name>  
**\_\_noregsave** <function specifier> <function name>  
<type specifier> **\_\_noregsave** <function name>

Description:

1. Functions declared by **#pragma regsave** and **\_\_regsave** generate codes that save and restore, at the entry and exit of the functions, the contents of all callee-save registers that should remain unchanged over a function call whether or not the registers are used in the function. In addition, register variables are not assigned to callee-save registers whose contents remain unchanged over a function call.
2. Functions declared by **#pragma noregsave** or **\_\_noregsave** do not generate codes for saving and restoring registers whether or not the registers are used by the function.
3. When a function declared by **#pragma noregsave** or **\_\_noregsave** is called, register variables are not assigned to registers whose contents should be retained after function call.

Example: (Compiled with CPU=2600a)

```
#pragma regsave (f,g) /* Declares generation of code */
                      /* for saving and restoring      */
                      /* register contents              */
#pragma interrupt g   /* Function g is an interrupt    */
                      /* function                      */
void f(void){}        /* Saves and restores ER2 to ER6 */
void g(void){}        /* Saves and restores ER0 to ER6 */
```

Remarks:

1. The first definition or declaration after the **#pragma regsave** or **#pragma noregsave** directive is treated as the target function.
2. Up to 63 functions can be declared in one **#pragma regsave/noregsave** directive.
3. A function call via a pointer-to-function is a standard function call even though an address of a function to which **\_\_noregsave** or **#pragma noregsave** is specified is assigned to that pointer. Hence the compiler may allocate a value to a callee-save register over the function call. The

value of the register may be changed by the call to the function with  
\_\_noregsave or #pragma noregsave.

Example:

```
#pragma noregsave f
void (*p)(void);
int sub(void)
{
    int a=8; // assume a is assigned to R4
    p=f;

    // R4 is saved before call below
    f();      // noregsave function call
             // R4 is restored after call above

    // R4 is NOT saved
    (*p)();   // standard function call
             // R4 is NOT restored
    return a;
}
```

**\_\_regparam2**

**\_\_regparam3**

Description Format: <type specifier> \_\_regparam2 <function name>  
                          <type specifier> \_\_regparam3 <function name>

Description: Specifies the number of parameter registers. Functions specified by  
\_\_regparam2 use the ER0 and ER1 registers (the R0 and R1 registers for  
the H8/300). Functions specified by \_\_regparam3 use the ER0, ER1, and  
ER2 registers (the R0, R1, and R2 registers for the H8/300).

Example:

```
void __regparam2 func1(long a, int b, int c, long d);
void __regparam3 func2(long a, int b, int c, long d);
int long a; int b; int c; long d;
```

```
void main(void)
{
    /* Variable allocation pattern*/
    : /* when cpu=2600a */
    func1(a, b, c, d); /* long a : ER0 */
    : /* int b : E1 */
    : /* int c : R1 */
    : /* long d : stack */
    :
```

```

func2(a, b, c, d);      /* long a : ER0          */
:                      /* int b  : E1          */
:                      /* int c  : R1          */
}                      /* long d : ER2          */

```

Remarks: This keyword cannot be specified prior to the <type specifier> and must be specified prior to the function name.

## #pragma option

Description Format: #pragma option [<option string>]

Description: Enables the options in the option string specified by **#pragma option**. This specification is valid until the file end is reached or until the point where **#pragma option** without <option string> is specified is reached.

If **#pragma option <keyword>** is specified, optimization specified by the keyword is performed. Table 10.25 lists the specifiable optimization options. For details on optimization options, refer to section 2, C/C++ Compiler Operating Method.

**Table 10.25 Specifiable Optimization Options**

Option Specification Method	Option Cancellation Method
case = {auto   ifthen   table}	None
Cmncode	nocmncode
Cpuexpand	nocpuexpand
Macsave	nomacsave
Regexpansion	noregexpansion
optimize	nooptimize
speed = {speed suboption}	None
sbr = {address}	None

Table 10.26 shows the speed sub-options.

**Table 10.26 Specifiable speed Options**

Option Specification Method	Option Cancellation Method
register	noregister
shift	noshift
loop	noloop
switch	noswitch
inline	noinline
struct	nostruct
expression	noexpression

When **#pragma option** without <option string> is specified, the previously-specified **#pragma option <option string>** is ignored and options specified on the command line become valid.

Example:

```
#pragma option speed
void func(void)           // speed option becomes valid
{
    :
}
#pragma option cpuexpand
void test(void)           // speed and cpuexpand become
                          // valid
{
    :
}
#pragma option           // Command line specification
void sub1(void)          // becomes valid
{
    :
}
```

Remarks:

**#pragma option=speed=inline=<value>** cannot be specified for H8SX and H8S (without **legacy=v4**). If **#pragma option speed=inline=<value>** for H8SX is attempted, the compiler assumes that **#pragma option speed=inline** is specified.



### (3) Other Extended Specifications

#### **#pragma asm**

Description Format: **#pragma asm**

<assembly-language instruction sequence>

**#pragma endasm**

Description: The assembly-language instructions must be preceded by **#pragma asm** and be followed by **#pragma endasm**.

The compiler expands the assembly-language instructions enclosed by **#pragma asm** and **#pragma endasm** into the object code generated by the compiler.

Example:

```
void func(void)
{
    #pragma asm
        CLRMAC          ; Clears the MAC register to 0
    #pragma endasm
    :
}
```

Remarks:

1. Specify assembly program output with the **code=asmcode** option when compiling. If not specified, the assembly-language instructions enclosed by **#pragma asm** and **#pragma endasm** are ignored.
2. The compiler checks neither the syntax of the assembly-language instructions, nor their influence over the code generated by the compiler. When the **optimize=1** or **speed** option is specified when compiling, the expanded code or location of the assembly-language instructions may differ from that specified using **#pragma asm** and **#pragma endasm**. Check the output code and program operation by yourself, when using this feature.
3. The **#pragma asm** and **#pragma endasm** specification cannot be nested. If attempted, an error will occur.
4. If **#pragma asm** and **#pragma endasm** are specified in a conditional or loop statement, the assembly-language instructions including **#pragma asm** and **#pragma endasm** must be enclosed by { }. If not, results are not guaranteed.
5. The assembler may display error message “402 ILLEGAL VALUE IN OPERAND”. This is the compiler generates the code without displacement. So be assembled it with optimize option. Or use the **JMP** instruction and modify the assembly-language program to satisfy the required branch width if necessary.

Example:

```
while(a==0)
{
    ..... Must always be specified
#pragma asm
    <assembly-language instruction string>
#pragma endasm
}
    ..... Must always be specified
```

## **\_\_asm**

Description Format: \_\_asm{

```
    ...
}
```

Description: Assembly-language instructions can be written in the range between \_\_asm { and }. This range is called an \_\_asm block afterwards. The language specification in the \_\_asm block is described below.

### 1. Syntax

- The compiler regards an \_\_asm block as a statement of C/C++ language. Though an \_\_asm block can be written where a statement can be written, the block cannot be written outside a function or before the declaration in a compound statement of C language.
- Up to one instruction can be written in one line.
- One instruction cannot be written across multiple lines.  
In the assembler, writing the '+' sign at a predetermined position allows to continue the current line to the next. In the \_\_asm block, however, the '+' sign is ineffective.
- A colon, ':' is necessary right after a label.  
The assembler regards a symbol beginning at the first column as a label. In the \_\_asm block, however, an instruction can be written from the first column. For the compiler to recognize a label in an \_\_asm block, a colon, ':' is necessary.
- A local label which begins with a '?' is not allowed.
- The comment in the C/C++ language format (/\* \*/ and // ) is allowed. The comment in the assembly language format ( ; ) is not allowed.
- Any comment in the \_\_asm block is not displayed in the assembly source output or in the object listing output.
- Except the .DATA directive, any assembly directives cannot be written. File inclusion, conditional assembly, macro and structured assembly are not supported.

## 2. Symbol

### 2-1 Variable name

- The name of a statically allocated variable is regarded as an address. The name of an **auto** variable is regarded as the displacement from the SP, the stack pointer. The prefix ‘\_’ appended to external variables by the compiler is not required in an `__asm` block. In the following example, x will be an absolute address, and y will be the displacement from the SP.

Example:

```
int x;
void func()
{
    int y;
    __asm {
        mov.w    @x,r0           //mov.w    @x,r0
        mov.w    @(y,sp),r1      //mov.w    @(0,sp),r1
    }
}
```

- C/C++ variables referred to from an `__asm` block will be allocated in the memory.
- **auto** variables and parameters of C++ cannot be referred to from an `__asm` block.

### 2-2 Function name

- Function names can be referred to from an `__asm` block if they have C linkage. The prefix ‘\_’ appended to external function name by the compiler is not required in an `__asm` block

### 2-3 Label

- Labels in C/C++ program cannot be referred to from an `__asm` block, and vice versa. A label in one `__asm` block cannot be referred to from another `__asm` block.
- Location counter, ‘\$’ can be used in an `__asm` block.

### 2-4 Enumerator name

- An enumerator name of enum type data can be used as a constant.

### 2-5 Struct member name

- “<struct variable name>.<memeber name>” will be an address if the variable is a statically allocated variable, or will be the offset from the SP if the variable is an **auto** variable.
- “OFFSET <struct variable name>.<memeber name>” or “OFFSET (<struct variable name>.<memeber name>)” will be the offset of the member from the top of the struct.
- The “->” operator used as “<struct variable name>-><memeber name>” or “OFFSET (<struct variable name>-><memeber name>)” is not allowed.

- Bit field cannot be written in an `__asm` block.

- Example:

```
struct S {
    int a;
    int b;
} x, *p;
void func()
{
    __asm {
        mov.w    @x.b,r0    // mov.w    @_x+2,r0
        mov.l    @p,er1     // mov.l    @_p,er1
        mov.w    r0,@(OFFSET(x.b),er1)
                                // mov.w    r0,@(2,er1)
    }
}
```

## 2-6 Section name

- A section name can be used only as an operand of `STARTOF` or `SIZEOF` operator.

## 3. Operator

- Operators of assembly language can be used in an `__asm` block. They are shown below.

unary plus: +, unary minus: -,

addition: +, subtraction: -, multiplication: \*, division: /,

unary not: ~, bit-wise and: &, bit-wise or: |, bit-wise exclusive or: ^,

arithmetic left shift: <<, arithmetic right shift: >>,

section start address: `STARTOF`, section size: `SIZEOF`

upper byte: `HIGH`, lower byte: `LOW`,

upper word: `HWORD`, lower word: `LWORD` .

## 4. Integer constant

- An integer constant can be specified in the C/C++ language format rather than in the assembly language format. For example, a hexadecimal number should be written as `0xFF` rather than `H'FF`.

## 5. Character constant

- A character constant can be specified in the C/C++ language format rather than in the assembly language format. For example, a character constant should be written as `'a'` rather than `"a"`. `"a"` is regarded as a string followed by a null character.

## 6. Register convention

- The register convention of an `__asm` block is similar to that of a function. Even though a caller-save register such as ER0, ER1 or (ER2) is used in an `__asm` block, the register is not saved or restored at the entry or exit of the block. If a callee-save register such as (ER2,) ER3, ER4, ER5 or ER6 is used in an `__asm` block, the compiler automatically generates code to save and restore the register at the entry and exit of the block, respectively. It is assumed that the SP is unchanged from the entry to the exit of an `__asm` block. After making a function call changes the SP, put back the SP to the original value before the call is made.
- Even though the MAC register is used in an `__asm` block, the compiler never generates code to save/restore the MAC register at the entrance/exit of the `__asm` block. When the MAC register is changed inside an `__asm` block and if the value of the MAC register should be preserved over the `__asm` block, add code to save/restore the MAC register in the `__asm` block. The compiler does not recognize that the MAC register is written even when the **macsave** option is specified to an interrupt function.

Example:

```
// -cpu=h8sxa
int g_x;
struct ST {
    int a;
    char b;
    char c;
} g_st;
enum color {BLUE, GREEN, YELLOW, RED};
/* Image of actual code */
void func(void) // Places local variables on the stack
{
    /* sub.w #6,r7 */
    int x;
    int y;
    struct ST l_st;
    // The __asm block saves the values of registers used
    // in func.
    __asm{
        /* stm.l (er2-er3),@-sp */
        // y : local, scalar, offset from SP = 8
        // l_st : local, struct, offset from SP = 10
        mov.w @(y,sp),r0 /* mov.w @(8,sp)r0 */
        mov.l #y,er1 /* mov.l #8,er1 */
        mov.w @(l_st.b, sp),r0 /* mov.w @(12,sp),r0 */
    }
```

```

        mov.l    #l_st.c,er1      /* mov.l #13,er1      */
        mov.l    #OFFSET l_st.c,er0 /* mov.l #3,er0      */
        mov.l    #l_st,er2        /* mov.l #10,er2     */
        bra      L1

CHAR:
        .data.b   'a'              /* .data.b H'61      */
STRING:
        .data.w   "abc"            /* .data.w H'6263    */
ENUM:
        .data.w   YELLOW           /* .data.w H'0002    */
BOTTOM:
        .data.l   STARTOF P + SIZEOF P
                                   /* .data.l H'00000000 */

L1:
        // g_st :
        // g_x  :
        mov.b     #0xFF,@g_st.b     /* mov.w #H'FF,@_g_st+2 */
        mov.l     #g_st.b,er1       /* mov.w #_g_st+2,er1   */
        mov.l     #OFFSET g_st.b,er2
                                   /* mov.w #2,er2        */
        mov.l     #g_st,er3         /* mov.w #_g_st,er3     */
        mov.w     #func,@g_x       /* mov.w #_func,@_g_x   */
        mov.l     #g_x,er0         /* mov.w #_g_x,er0      */
    }                               /* The contents of registers used in the __asm
                                   // block have been restored
                                   /* ldm.l @sp+,(er2-er3) */
}

```

Remarks:

1. This keyword is valid only when the CPU type is H8SX or H8S.
2. The assembly program written in the `__asm` block can be compiled into an object file directly with the **code=machine** option.
3. If the SP is changed in the `__asm` block, the source-level debugging is not guaranteed.
4. The assembler may display error message “402 ILLEGAL VALUE IN OPERAND”. This is the compiler generates the code without 3. displacement. So be assembled it with optimize option. Or use the **JMP** instruction and modify the assembly-language program to satisfy the 3. required branch width if necessary.

## **#pragma global\_register** **\_ \_global\_register**

Description Format: `#pragma global_register (<variable name>=<register name>[,...])`  
`_ _global_register(<register name>) <type specifier> <variable name>`  
`<type specifier> _ _global_register(<register name>) <variable name>`  
`<variable name>`: Local variable and C++ non-static member data cannot be specified  
`<register name>`: ER4, ER5 (R4, R5 for H8/300)

Description: Allocates the global variable specified in `<variable name>` to the register specified in `<register name>`.

Example:

```
#pragma global_register(x=R4)    /* External variable */
                                /* x is allocated to */
                                /* R4                */

int x;

_ _global_register(R5L) char y; /* External variable */
                                /* y is allocated to */
                                /* R5L               */

void func1(void)
{
    x++;
}

void func2(void)
{
    y=0;
}

void func(int  a)
{
    x = a;
    func1();
    func2();
}
```

Remarks:

1. The variables defined and declared after the **#pragma global\_register** are the target variables.
2. This function is used for a simple or pointer type variable in the global variable. Do not specify a **double** type variable.
3. The initial value cannot be set. In addition, the address cannot be referenced.

4. The specified variable cannot be referenced from the linked file which does not have the register specification.
5. Setting and reference in the interrupt functions are not guaranteed.
6. The duplication of the specification of the same variable or register is prohibited. You can not specify the variables which are specified with **#pragma abs8**, **#pragma abs16**, **\_\_abs8**, **\_\_abs16**, **\_\_near8**, or **\_\_near16**.

**#pragma pack 1**

**#pragma pack 2**

**#pragma unpack**

Description Format: **#pragma pack 1**  
**#pragma pack 2**  
**#pragma unpack**

Description: Specifies the boundary alignment for structure, unions, and class members after the **#pragma pack 1** or **#pragma pack 2** is specified in the source program. The boundary alignment value specified by the **pack** option is used for structures, unions, and class members declared when **#pragma pack 1** or **#pragma pack 2** has not been specified or after **#pragma unpack** has been specified. Table 10.27 shows the boundary alignment specified by **#pragma pack 1**, **#pragma pack 2**, and **#pragma unpack**.

**Table 10.27 Boundary Alignment of Structures, Unions, and Class Members**

Extension/ Member Type	<b>#pragma pack 1</b>	<b>#pragma pack 2</b>	<b>#pragma unpack (or No Extension Specified)</b>
[unsigned] char	1	1	1
[unsigned] short, [unsigned] int, [unsigned] long, floating-point number, pointer type	1	2	Value specified by <b>pack</b> option
Structures, unions, and classes aligned to one-byte boundary	1	1	1
Structures, unions, and classes aligned to two-byte boundary	1	2	Value specified by <b>pack</b> option



```

Example:  #pragma pack 2
          struct S1 {
              char a;          /* offset: 0          */
                                /* gap: 1 byte      */
              int b;          /* offset: 2          */
              char c;          /* offset: 4          */
                                /* gap: 1 byte      */
          };
          #pragma pack 1
          struct S2 {
              char a;          /* offset: 0          */
              int b;          /* offset: 1          */
              char c;          /* offset: 3          */
          };
          #pragma unpack      /* Follows pack option. Assumes */
                                /* pack=2 as default.          */
          struct S3 {
              char a;          /* offset: 0          */
                                /* gap: 1 byte      */
              int b;          /* offset: 2          */
              char c;          /* offset: 4          */
                                /* gap: 1 byte      */
          };
          struct S1 s1 = {1,2,3}; /* _s1: .data.b 1,0,0,2,3,0 */
          struct S2 s2 = {1,2,3}; /* _s2: .data.b 1,0,2,3 */
          struct S3 s3 = {1,2,3}; /* _s3: .data.b 1,0,0,2,3,0 */

          void test()          /* _test:          */
          {                    /* mov.w #1,R0      */
              s1.b=1;          /* mov.w R0,@_s1+2  */
              s2.b=2;          /* mov.w #2,R0;For members */
              :                /* mov.b R0H,@_s2+1; aligned to */
                                /* ; one-byte      */
                                /* ; boundary,     */
                                /* mov.b R0L,@_s2+2; Settings */
                                /* ; and          */
                                /* ; references    */
                                /* ; are done in   */
                                /* ; one-byte      */
                                /* ; units         */
          }

```

- Remarks:
1. The boundary alignment for structure members can be specified also by the **pack** option. When the option, and the **#pragma pack 1** or **2** are specified together, the **#pragma pack 1** or **2** takes priority.
  2. The boundary alignment for structures, unions, and classes equals to the maximum boundary alignment for the members. For details, refer to section 10.1.2, Internal Data Representation, (2) Compound Type (C), Class Type (C++).
  3. A member of a struct, union or class to which **#pragma pack 1** or the **pack=1** option is specified must not be accessed via a pointer

(including an access via a pointer in a member function).

Example: (cpu=2600a)

```
struct S {
    char x;
    int y;
} s;
int *p=&s.y; // address of s.y can be odd
void test()
{
    s.y=1;    // accessed correctly
    *p =1;    // can be accessed incorrectly
}
```

## **\_\_evenaccess**

Description Format: `__evenaccess <type specifier> <variable name>`  
`<type specifier> __evenaccess <variable name>`

Description: Ensures access to an integer-type variable to be done within the size of the declared variable type.  
For the H8/300, 4-byte scalar-type variables are accessed in 2-byte units.  
For the H8SX, refer to the remarks below.

Example: 

```
#define A (*(volatile unsigned short __evenaccess
*)0xff0178)
void test(void)
{
    A &= ~0x2000 ;
}
```

When **\_\_evenaccess** is not specified  
(1-byte memory access by BCLR.B)

```
_test:
    MOV.L    #H'FF0178,ER0
    BCLR.B   #H'5,@ER0
    RTS
```

When **\_\_evenaccess** is specified  
(2-byte memory access by MOV.W)

```
_test:
    MOV.W    @H'FF0178:24,R0
    BCLR.B   #H'5,R0H
    MOV.W    R0,@H'FF0178:24
    RTS
```

Remarks:

- If a 2-byte counter register is accessed in 1-byte unit, the 1 byte that is not accessed may have an incorrect value. In this case, specify **\_\_evenaccess** to the counter register to access it with the correct size.
- When the CPU is H8SX, **\_\_evenaccess** can be specified for all types and the member including bit fields of structures, unions, and classes. When **\_\_evenaccess** is specified for structures, unions, and classes, access is the same as that when specified for each member.
- The double type cannot be accessed in 8-byte units.
- When the little-endian space is supported by H8SX, access a datum in the size of its type using **\_\_evenaccess**.
- In H8SX, an error will occur if the initial value is specified for the static variable with **\_\_evenaccess** declaration in order to avoid placing the initial value of big endian in the little-endian space.

Example:

```
__evenaccess long x=0x12345678;      /* Error */
void f (void)
{
    ...
}
```

- Structures cannot be used in simple assignment, as parameters, or as return values when the CPU setting is H8SXN/H8SXM/H8SXA/H8SXX and keyword **evenaccess** has been specified. In these cases, only member-by-member setting and reference are possible.

Example:

```
typedef struct {
    int a;
    long b;
}str;

__evenaccess str st1;
str st2;
void func(str);

str main(void){
    str temporary;
    st2.a = st1.a;      /* For a structure declared as      */
    st2.b = st2.b;      /* __evenaccess, simple assignment */
                        /* is achieved through member-by  */
                        /* -member operations.           */

    temporary.a = st1.a; /* Member-by-member assignment */
    temporary.b = st1.b; /* of the structure declared   */
    func(temporary);     /* as __evenaccess to a       */
                        /* structure not declared as  */
                        /* __evenaccess allows        */
                        /* specification of the latter */
                        /* as a parameter.           */
}
```

```

return (temporary);                                /* The structure not declared */
                                                    /* as __evenaccess is usable */
                                                    /* as a return value.        */
}

```

## #pragma address

**Description Format:**      `#pragma address (<variable name>=<absolute address> [...])`  
                                  `<absolute address> : Effective address`  
                                  (in hexadecimal notation of the C language)

**Description:**      For linkage, the compiler allocates a single specified variable to `<absolute address>` by setting up the section to which the variable is allocated at `<absolute address>`. When consecutive addresses are specified for variables of the same section type, the compiler places them in the same section. If a variable is allocated to an address within the 8-bit or 16-bit short absolute area, the compiler outputs 8- and 16-bit absolute instructions (forms with :8 or :16), except in cases where a variable requiring even boundaries is specified for an 8-bit short absolute area.

**Example:**              (1)

- Source program

```

#pragma address (io=0x100)
int io;

func(void){
    io = 0;
}

```

- Output object

When **#pragma address** is not specified

```

.SECTION P, CODE
_func:
    MOV.W  #0:4, @_io:32
    RTS
.SECTION B, DATA, ALIGN=2
_io:
    .RES.W  1
    .END

```

When **#pragma address** is specified

```
.SECTION P, CODE
_func:
    MOV.W    #0, @_io:16
    RTS
    .SECTION $ADDRESS$B100, DATA, LOCATE=H'100
_io:
    .RES.W 1
    .END
```

(2)

- Source program

```
#pragma address (P1=0x100)
struct {
    unsigned char  BYTE;
    unsigned short WORD;
}P1;

func()
{
    P1.WORD =10;
}
```

- Output object

When **#pragma address** is not specified

```
.SECTION P, CODE
_func:
    MOV.W    #10, @_P1+2:32
    RTS
    .SECTION B, DATA, ALIGN=2
_P1:
    .RES.W 2
    .END
```

When **#pragma address** is specified

```
.SECTION P, CODE
_func:
    MOV.W    #10, @_P1+2:16
    RTS
    .SECTION $ADDRESS$B100, DATA, LOCATE=H'100
_P1:
    .RES.W   2
    .END
```

(3) Variables at consecutive addresses have the same section type

- Source program

```
#pragma address (io=0x100, io2=0x102)
int io;
int io2;

func(void){
    io  =0;
    io2 =0;
}
```

- Output object

When **#pragma address** is not specified

```
.SECTION P, CODE
_func:
    MOV.W    #0:4, @_io2:32
    MOV.W    #0:4, @_io:32
    rts
    .SECTION B, DATA, ALIGN=2
_io:
    .RES.W   1
_io2:
    .RES.W   1
    .END
```

When **#pragma address** is specified

```
.SECTION P, CODE
_func:
    MOV.W    #0, @_io2:16
    MOV.W    #0, @_io:16
    RTS
    .SECTION $ADDRESS$B100, DATA, LOCATE=H'100
_io:
    .RES.W 1
_io2:
    .RES.W 1
.END
```

(4) Variables have the same section type but are not consecutive  
(in the example below, this leaves two bytes empty).

- Source program

```
#pragma address (io=0x100, io2=0x104)
int io;
int io2;

func(void){
    io = io2 = 0;
}
```

- Output object

When **#pragma address** is not specified

```
.SECTION P, CODE
_func:
    MOV.W    #0:4, @_io2:32
    MOV.W    #0:4, @_io:32
    RTS
    .SECTION B, DATA, ALIGN=2
_io:
    .RES.W 1

_io2:
    .RES.W 1
.END
```



When **#pragma address** is specified

```
.SECTION P, CODE
_func:
    MOV.W    #0, @_io2:16
    MOV.W    #0, @_io:16
    RTS
    .SECTION $ADDRESS$B100, DATA, LOCATE=H'100
_io:
    .RES.W 1
    .SECTION $ADDRESS$B104, DATA, LOCATE=H'104
_io2:
    .RES.W 1
    .END
```

Remarks:

- This function is only valid when the CPU type is H8SX or H8S.
- For a given variable, **#pragma address** must be specified before the variable is declared.
- An error occurs if a compound/class-type member, static member, or symbolic name other than that of a variable is specified.
- An error occurs if an odd address is specified for a variable or structure requiring alignment with an even boundary.
- An error occurs if more than one **#pragma address** specification is made for the same variable.
- An error occurs if the same address is specified for different variables or the addresses of variables overlap.
- An error occurs if more than one of the following **#pragma** extensions is specified for the same variable.
  - #pragma section**
  - #pragma abs8/abs16**
  - #pragma global\_register**
- Do not specify **#pragma address** for a variable initialized with data. If you do make such a specification, the compiler outputs message C1407 (W) **#pragma address** ignored.

## 10.2.2 Section Address Operator

**`__sectop`**

**`__secend`**

Description Format: `__sectop("<section name>")`  
`__secend("<section name>")`

Description: Refers to the start address of <section name> specified by `__sectop`.  
Refers to the end + 1 address of <section name> specified by `__secend`.

Example:

```
#include <machine.h>
#pragma section $DSEC
static const struct {
    void *rom_s;          /* Start address of initialized */
                          /* data section in ROM          */
    void *rom_e;          /* End address of initialized   */
                          /* data section in ROM          */
    void *ram_s;          /* Start address of initialized */
                          /* data section in RAM          */
}DTBL[]={__sectop ("D"), __second ("D"), __sectop ("R")};

#pragma section $BSEC
static const struct {
    void *b_s;            /* Start address of uninitialized */
                          /* data section                    */
    void *b_e;            /* End address of uninitialized   */
                          /* data section                    */
}BTBL[]={__sectop ("B"), __second ("B")};

#pragma section
#pragma stacksize 0x100 /* Declares stack section S */
#pragma entry INIT      /* Declares function INIT as */
                          /* an entry function         */
void main(void);        /* Declares main function */
void INIT(void)          /* _INIT: Entry start function */
{
    /* MOV #STARTOF S+SIZEOF S,SP */
    /*                               ; SP initial */
    /*                               ; settings  */
    _INITSCT();           /* JSR @_ _INITSCT ; Initializes */
    /*                               ; section area*/
    main();               /* JSR @_main      ; Calls main */
    /*                               ; function   */
    sleep();              /* SLEEP          ; Sleep state */
    /*                               ; in low-power*/
    /*                               ; consumption */
    /*                               ; mode       */
}


```

For details of section initialization, refer to section 9.2.2, Execution Environment Settings.

### 10.2.3 Intrinsic Functions

The compiler provides the following functions that cannot be written in C/C++, as intrinsic functions.

- Setting and referencing the condition code register
- Setting and referencing the extend register
- Multiply and accumulate (MAC) operation
- Rotation
- Special instructions (TRAPA, SLEEP, MOVFPE, MOVTPE, EEPMOV, TAS, NOP, and XCH)
- Overflow testing
- Decimal operation

Intrinsic functions can be written in the same call format as regular functions. However, when using intrinsic functions, **#include <machine.h>** must be declared.

Table 10.28 lists intrinsic functions.

**Table 10.28 Intrinsic Functions**

Item	Specification	Function
Condition code register	void set_imask_ccr(unsigned char mask)	Sets value of parameter mask in the interrupt mask
	unsigned char get_imask_ccr(void)	References the interrupt mask
	void set_ccr(unsigned char ccr)	Sets the condition code register (value of parameter ccr -> CCR)
	unsigned char get_ccr(void)	References the condition code register
	void and_ccr(unsigned char ccr)	ANDs the condition code register (CCR & parameter ccr -> CCR)
	void or_ccr(unsigned char ccr)	ORs the condition code register (CCR   parameter ccr -> CCR)
	void xor_ccr(unsigned char ccr)	Exclusively ORs the condition code register (CCR ^ parameter ccr -> CCR)

**Table 10.28 Intrinsic Functions (cont)**

Item	Specification	Function
Extend register	void set_imask_exr(unsigned char mask)	Sets the value of parameter mask in the interrupt mask
	unsigned char get_imask_exr (void)	References the interrupt mask
	void set_exr(unsigned char exr)	Sets the extend register (parameter exr -> EXR)
	unsigned char get_exr (void)	References the extend register
	void and_exr(unsigned char exr)	ANDs the extend register (EXR & parameter exr -> EXR)
	void or_exr(unsigned char exr)	ORs the extend register (EXR   parameter exr -> EXR)
	void xor_exr(unsigned char exr)	Exclusively ORs the extend register (EXR ^ parameter exr -> EXR)
Vector base register	void set_vbr(void* vbr)	Makes the VBR setting
Multiply and accumulate operation	long mac (long val,int* ptr1, int* ptr2,unsigned long count)	Calculates $val+\sum_{i=0}^{count-1} (ptr1[i]*ptr2[i])$ using MAC instruction, or calculates $val+\sum_{i=0}^{count-1} (ptr1[i]*((ptr2+i)\&mask))$ using ring buffer function
	long mac1 (long val,int* ptr1, int* ptr2,unsigned long count, unsigned long mask)	
64-bit multiplication	long mulsu (long val1, long val2)	Expanded to MULS/U instruction
	unsigned long muluu (unsigned long val1, unsigned long val2)	Expanded to MULU/U instruction
Rotation	char rotlc(int count,char data)	Rotates data to the left for the number of bits specified in count
	int rotlw(int count,int data)	
	long rotll(int count,long data)	
	char rotrc(int count,char data)	Rotates data to the right for the number of bits specified in count
	int rotrw(int count,int data)	
	long rotrl(int count,long data)	

**Table 10.28 Intrinsic Functions (cont)**

Item	Specification	Function
Special instructions	void trapa(unsigned int trap_no)	Expanded to TRAPA #trap_no
	void sleep(void)	Expanded to SLEEP instruction
	void movfpe(char *addr,char data) char _movfpe(char *addr)	Sets *addr in data using MOVFPE instruction, or returns *addr
	void movtpe(char data,char *addr)	Sets data in *addr using MOVTPE instruction
	void tas(char *addr)	Compares *addr with 0, sets the results in the condition code register, and sets most significant bit of *addr as 1 by using the TAS instruction
	void eepmov(void *dst,const void *src, unsigned char size)	Transfers data for bytes specified in size from *src to *dst by using EEPMOV instruction
	void eepmov(void *dst,const void *src, unsigned int size)	
	void eepmovb (void *dst,const void *src, unsigned char size)	
	void eepmovw (void *dst,const void *src, unsigned int size)	
	void eepmovi (void *dst, const void *src, unsigned char size)	Transfers data from *src to *dst for the number of times specified by count by using movmd.b instruction
	void eepmovi (void *dst, void *src, unsigned int size)	
	void movmdb (void *dst, const void *src unsigned int count)	
	void movmdw (int *dst, const int *src, unsigned int count)	
	void movmdl (long *dst, const long *src, unsigned int count)	Transfers data from *src to *dst for the number of times specified by count by using movmd.l instruction
	unsigned int movsd (char *dst, const char *src unsigned int size)	Transfers data up to the maximum number of bytes specified by size. However, transferring a zero datum terminates execution.
	void nop(void)	Expanded to NOP instruction

**Table 10.28 Intrinsic Functions (cont)**

Item	Specification	Function
Condition code operation	int ovfaddc(char dst,char src,char *rst)	Sets the results of dst + src in *rst and reflects the results in the condition code register
	int ovfadduc(unsigned char dst, unsigned char src,unsigned char *rst)	
	int ovfaddw(int dst,int src,int *rst)	
	int ovfadduw(unsigned int dst, unsigned int src,unsigned int *rst)	
	int ovfaddl(long dst,long src,long *rst)	
	int ovfaddul(unsigned long dst, unsigned long src,unsigned long *rst)	
	int ovfsubc(char dst,char src,char *rst)	Set the results of dst – src in *rst and reflects the results in the condition code register
	int ovfsubuc(unsigned char dst, unsigned char src,unsigned char *rst)	
	int ovfsubw(int dst,int src,int *rst)	
	int ovfsubuw(unsigned int dst, unsigned int src,unsigned int *rst)	
	int ovfsubl(long dst,long src,long *rst)	
	int ovfsubul(unsigned long dst, unsigned long src,unsigned long *rst)	
Condition code operation	int ovfshalc(char des,char *rst)	Sets the results of dst << 1 in *rst and reflects the results in the condition code register (arithmetical shift)
	int ovfshalw(int dst,int *rst)	
	int ovfshall(long dst,long *rst)	
	int ovfshlluc(unsigned char des, unsigned char *rst)	Sets the results of dst << 1 in *rst and reflects the results in the condition code register (logical shift)
	int ovfshlluw(unsigned int dst, unsigned int *rst)	
	int ovfshllul(unsigned long dst, unsigned long *rst)	
	int ovfnegc(char dst,char *rst)	Sets the 2's complement of dst in *rst and reflects the results in the condition code register
	int ovfnegw(int dst,int *rst)	
	int ovfnegl(long dst,long *rst)	



**Table 10.28 Intrinsic Functions (cont)**

Item	Specification	Function
Decimal operation	void dadd(unsigned char size, char *ptr1,char *ptr2,char *rst)	Assumes ptr1 and ptr2 to be decimal arrays consisting of digits indicated in size, performs addition in decimals, and sets the results in *rst
	void dsub(unsigned char size, char *ptr1,char *ptr2,char *rst)	Assumes ptr1 and ptr2 to be decimal array consisting of digits indicated in size, performs subtraction in decimals, and sets results in *rst

**void set\_imask\_ccr(unsigned char mask)**

Description: Sets the value of parameter **mask** (0 or 1) to the interrupt mask bit (I) of the condition code register (CCR).

Header: <machine.h>

Parameters: mask            Mask value (0 or 1)

```
Example: #include <machine.h>      /* Must include <machine.h> */
void main(void)
{
    set_imask_ccr(0); /* Clears interrupt mask bit */
}
```

**unsigned char get\_imask\_ccr(void)**

Description: References the value of parameter **mask** (0 or 1) of the interrupt mask bit (I) of the condition code register (CCR).

Header: <machine.h>

Return value: Reference value of the interrupt mask bit of the condition code register (CCR)

```
Example: #include <machine.h>      /* Must include <machine.h> */
void main(void)
{
    if(get_imask_ccr()) /*Refers to interrupt mask bit*/
    :
}
```

## **void set\_ccr(unsigned char ccr)**

**Description:** Sets the value of parameter **ccr** (8 bits) to the condition code register (CCR).

**Header:** <machine.h>

**Parameters:** ccr                      Setting value (8 bits)

**Example:**

```
#include <machine.h>      /* Must include <machine.h> */
main()
{
    set_ccr(0);            /* Clears CCR                */
}
```

## **unsigned char get\_ccr(void)**

**Description:** References the value of the condition code register (CCR).

**Header:** <machine.h>

**Return value:** Reference value of the condition code register (CCR)

**Example:**

```
#include <machine.h>      /* Must include <machine.h> */
void main(void)
{
    unsigned char a;
    a=get_ccr();           /* Refers to CCR                */
}
```

## **void and\_ccr(unsigned char ccr)**

**Description:** ANDs the condition code register (CCR) with the value of parameter **ccr** and stores the results in the CCR.

**Header:** <machine.h>

**Parameters:** ccr                      Operand of logical AND operation

**Example:**

```
#include <machine.h> /* Must include <machine.h> */
void main(void)
{
    and_ccr(0x10);    /* Sets CCR & 0x10 in CCR */
}
```

## **void or\_ccr(unsigned char ccr)**

**Description:** ORs the condition code register (CCR) with the value of parameter **ccr** and stores the results in the CCR.

**Header:** <machine.h>

**Parameters:** ccr                      Operand of logical OR operation

**Example:**

```
#include <machine.h> /* Must include <machine.h> */
void main(void)
{
    or_ccr(0x10);     /* Sets CCR | 0x10 in CCR */
}
```

## **void xor\_ccr(unsigned char ccr)**

**Description:** Exclusively ORs the condition code register (CCR) with the value of parameter **ccr** and stores the results in the CCR.

**Header:** <machine.h>

**Parameters:** ccr                      Operand of logical exclusive OR operation

**Example:**

```
#include <machine.h>    /* Must include <machine.h> */
void main(void)
{
    xor_ccr(0x10);      /* Sets CCR ^ 0x10 in CCR */
}
```

## **void set\_imask\_exr(unsigned char mask)**

**Description:** Sets the value of parameter **mask** (0 to 7) to the interrupt mask bits (I2 to I0) of the extend register (EXR). This function can be used in **H8SXN**, **H8SXM**, **H8SXA**, **H8SXX**, **2600a**, **2000a**, **2600n**, and **2000n** CPU/operating modes.

**Header:** <machine.h>

**Parameters:** mask                      Mask value

**Example:**

```
#include <machine.h>    /* Must include <machine.h> */
void main(void)
{
    set_imask_exr(0);   /* Sets mask level 0 to the */
                       /* interrupt mask bits in the */
                       /* extended register */
                       :
}
```

## **unsigned char get\_imask\_exr(void)**

**Description:** References the value (0 to 7) of the interrupt mask bits (I2 to I0) of the extend register (EXR). This function can be used in **H8SXN, H8SXM, H8SXA, H8SXX, 2600a, 2000a, 2600n, and 2000n** CPU/operating modes.

**Header:** <machine.h>

**Return value:** Reference value of the interrupt mask bits of the extended register (EXR)

**Example:**

```
#include <machine.h>          /* Must include <machine.h> */
void main(void)
{
    if(get_imask_exr()); /* Refers to the interrupt */
                        /* mask bits of the extended */
                        /* register */
}
```

## **void set\_exr(unsigned char exr)**

**Description:** Sets the value of parameter **exr** (8 bits) to the extend register (EXR). This function can be used in **H8SXN, H8SXM, H8SXA, H8SXX, 2600a, 2000a, 2600n, and 2000n** CPU/operating modes.

**Header:** <machine.h>

**Parameters:**      **exr**              Setting value

**Example:**

```
#include <machine.h> /* Must include <machine.h> */
void main(void)
{
    set_exr(0); /* Clears the extended register */
}
```

## **unsigned char get\_exr(void)**

**Description:** References the extend register (EXR). This function can be used in **H8SXN, H8SXM, H8SXA, H8SXX, 2600a, 2000a, 2600n, and 2000n** CPU/operating modes.

**Header:** <machine.h>

**Parameters:** Reference value of the extended register (EXR)

**Example:**

```
#include <machine.h> /* Must include <machine.h> */
void main(void)
{
    unsigned char a;
    a=get_exr();      /* Refers to the extended register */
    :
}
```

## **void and\_exr(unsigned char exr)**

**Description:** ANDs the extend register (EXR) with the value of parameter **exr** and stores the result in the EXR. This function can be used in **H8SXN, H8SXM, H8SXA, H8SXX, 2600a, 2000a, 2600n, and 2000n** CPU/operating modes.

**Header:** <machine.h>

**Parameters:** **exr**                      Operand of logical AND operation

**Example:**

```
#include <machine.h> /* Must include <machine.h> */
void main(void)
{
    and_exr(0x10);    /* Sets EXR & 0x10 in EXR */
}
```

## **void or\_exr(unsigned char exr)**

**Description:** ORs the extend register (EXR) with the value of parameter **exr** and stores the result in the EXR. This function can be used in **H8SXN, H8SXM, H8SXA, H8SXX, 2600a, 2000a, 2600n, and 2000n** CPU/operating modes.

**Header:** <machine.h>

**Parameters:** exr                      Operand of logical OR operation

**Example:**

```
#include <machine.h>       /* Must include <machine.h>       */
void main(void)
{
    or_exr(0x10);           /* Sets EXR | 0x10 in EXR       */
}
```

## **void xor\_exr(unsigned char exr)**

**Description:** Exclusively ORs the extend register (EXR) with the value of parameter **exr** and stores the result in the EXR. This function can be used in **H8SXN, H8SXM, H8SXA, H8SXX, 2600a, 2000a, 2600n, and 2000n** CPU/operating modes.

**Header:** <machine.h>

**Parameters:** exr                      Operand of logical exclusive OR operation

**Example:**

```
#include <machine.h>       /* Must include <machine.h>       */
void main(void)
{
    xor_exr(0x10);       /* Sets EXR ^ 0x10 in EXR       */
}
```

## **void set\_vbr(void\* vbr)**

**Description:** Sets **vbr** (32 bits) to the vector base register (VBR). This function can be used when the CPU type is **H8SXN**, **H8SXM**, **H8SXA**, or **H8SXX**.

**Header:** <machine.h>

**Parameters:** **vbr**                      Setting value

**Example:**

```
#include <machine.h> /* Be sure to include <machine.h> */
void main(void)
{
    set_vbr((void*)0x20000);    /* Sets 0x20000 to VBR */
}
```

## **long mac(long val,int \*ptr1,int \*ptr2,unsigned long count)** **long macl(long val,int \*ptr1,int \*ptr2,unsigned long count,unsigned long mask)**

**Description:** Expanded to the multiply-and-accumulate instruction, MAC.  
The function **mac** sets parameter **val** to the MAC register as the initial value, multiplies two bytes **ptr1** and **ptr2** with sign, adds the 4-byte result to the MAC register contents, and adds two to **ptr1** and **ptr2**. This operation is repeated for the number of times specified by **count**.  
The **macl** function logically ANDs the values of **ptr2** and **mask** so that the data of **ptr2** can be used as a ring buffer.  
These functions can be used in **H8SXN**:{**M** | **MD**}, **H8SXM**:{**M** | **MD**}, **H8SXA**:{**M** | **MD**}, **H8SXX**:{**M** | **MD**}, **2600a** and **2600n** CPU/operating modes.

**Header:** <machine.h>

**Return value:** Result of multiply-and-accumulate operation

**Parameters:**

<b>val</b>	Initial value of the MAC register
<b>ptr1, ptr2</b>	Pointer to the multiplication data
<b>count</b>	Number of loops
<b>mask</b>	Mask value for the ring buffer



Example:

```
#include <machine.h>          /* Must include<machine.h>*/
int ptr1[10]={0,1,2,3,4,5,6,7,8,9};
int ptr2[10]={9,8,7,6,5,4,3,2,1,0};
long l1,l2;

:
void main(void)
{
    l1=mac(100,ptr1,ptr2,4);    /* Executes          */
                                /* 11=100+0*9+1*8+2*7+3*6 */
    l2=macl(100,ptr1,ptr2,4,-4); /* Executes          */
                                /* 12=100+0*9+1*8+2*9+3*8 */
                                /* The data of ptr2[0] and */
                                /* ptr2[1] is repeatedly used*/
                                /* as a ring buffer. Since */
                                /* ptr2 & mask is used as an */
                                /* address, ptr2 must be */
                                /* assigned to an address */
                                /* that is a multiple of */
                                /* eight.          */
}
```

Remarks:

The boundary of the table pointed to by **ptr2** in the **macl** function must be aligned to a double of the **mask** value's complement. For example, in the case above, the linkage map must be confirmed so that **ptr2** is allocated to the address of a multiple of eight.

**long mulsu (long val1, long val2)**

**unsigned long muluu (unsigned long val1, unsigned long val2)**

**Description:** Expanded to the muls/u or mulu/u instruction, which performs 32-bit x 32-bit = 64-bit multiplication.

32-bit parameters (**val1** and **val2**) for this intrinsic function are multiplied and the upper 32 bits are returned as the operation result.

**Header:** <machine.h>

**Parameters:**

val1	Multiplicand
val2	Multiplier

**Example:**

```
#include <machine.h>
long      s_val1, s_val2, s_ans;
unsigned long  u_val1, u_val2, u_ans;
void f(void)
{
    s_ans = mulsu (s_val1, s_val2);
           /*Signed 32-bit multiplication*/

    u_ans = muluu (u_val1, u_val2);
           /*Unsigned 32-bit multiplication*/
}
```

**Remarks:** This intrinsic function is only valid when the CPU is H8SXN:{M | MD}, H8SXM:{M | MD}, H8SXA:{M | MD} or H8SXX:{M | MD}.

**char rotlc (int count,char data)**

**int rotlw (int count,int data)**

**long rotll (int count,long data)**

Description: Functions **rotlc**, **rotlw**, and **rotll** rotate 1-byte, 2-byte, and 4-byte data to the left by the number of bits specified by **count**, respectively, and return the results.

Header: <machine.h>

Return value: Result of data rotation

Parameters:      count              Number of bits to be rotated  
                 data                Data to be rotated

Example:            

```
#include <machine.h>      /* Must include <machine.h>      */
int i,data;
void f(void)
{
    i=rotlw(5,data);   /* Rotates data 5 bits to the left */
}
```

**char rotrc (int count,char data)**

**int rotrw (int count,int data)**

**long rotrl (int count,long data)**

Description: The functions **rotrc**, **rotrw**, and **rotrl** rotate 1-byte, 2-byte, and 4-byte data to the right by the number of bits specified by **count**, respectively, and return the results.

Header: <machine.h>

Return value: Result of data rotation

Parameters:      count              Number of bits to be rotated  
                 data                Data to be rotated

Example:            

```
#include <machine.h>      /* Must include <machine.h>      */
int i,data;
void f(void)
{
    i=rotrw(5,data);   /* Rotates data 5 bits to the right*/
}
```

## **void trapa(unsigned int trap\_no)**

**Description:** Expanded to an unconditional trap instruction, TRAPA #**trap\_no**. The **trap\_no** must be a constant from 0 to 3. This function cannot be used in **300** CPU/operating mode.

**Header:** <machine.h>

**Parameters:** trap\_no      Trap number for the vector address indicating the jump destination

**Example:**

```
#include <machine.h>  /* Must include <machine.h> */
void f(void)
{
    :
    trapa(0);           /* Returns at trapa #0      */
}
```

## **void sleep(void)**

**Description:** Expanded to a low-power consumption instruction, SLEEP.

**Header:** <machine.h>

**Example:**

```
#include <machine.h>  /* Must include <machine.h>      */
void f(void)
{
    :
    sleep();           /* Expanded to a sleep      */
                       /* instruction               */
}
```

**void movfpe(char \*addr,char data)**  
**char \_movfpe (char \*addr)**

**Description:** The contents of **\*addr** is moved to **data** by the function **movfpe** or is returned by the function **\_movfpe** at a timing synchronous to the E clock using the E clock-synchronous data transfer instruction, MOVFPE. For **\*addr**, specify a value that can be accessed by a 16-bit absolute address.

**Header:** <machine.h>

**Return value:** The **movfpe** function N/A  
The **\_movfpe** function Destination data

**Parameters:** addr Pointer to the source data  
data Destination data (the **movfpe** function)

**Example:**

```
#include <machine.h> /* Must include <machine.h> */
#pragma abs16 a /* Declares the first parameter */
char a,data; /* by #pragma abs16 to access it */
/* by a 16-bit absolute address */

void f(void)
{
    movfpe(&a,data); /* Moves a to data by MOVFPE*/
    data = _movfpe(&a); /* Same operation as the above.*/
}
```

**Remarks:** char \_movfpe(char \* addr) is valid only with H8SX.

## **void movtpe(char data,char \*addr)**

**Description:** Moves the contents of **data** to **\*addr** at a timing synchronous to the E clock using the E clock-synchronous data transfer instruction, MOVTPPE. For **\*addr**, specify a value that can be accessed by a 16-bit absolute address.

**Header:** <machine.h>

**Parameters:**      data              Source data  
                     addr              Pointer to the destination

**Example:**

```
#include <machine.h>    /* Must include <machine.h>      */
#pragma abs16 a          /* Declares the second parameter*/
char a,data;            /* by #pragma abs16 to access it*/
                        /* by a 16-bit absolute address */

void f(void)
{
    movtpe(data,&a);     /* Moves data to a at a timing */
                        /* synchronous to the E clock */
}
```

## **void tas(char \*addr)**

**Description:** Expanded to a test and set instruction, TAS. Compares the contents of **addr** with 0, reflects the result in the condition code register (CCR), and changes the highest-order bit of the **addr** contents to 1. This function can be used in **H8SXN**, **H8SXM**, **H8SXA**, **H8SXX**, **2600a**, **2000a**, **2600n**, and **2000n** CPU/operating modes.

**Header:** <machine.h>

**Parameters:**      addr              Pointer to the data to be tested and set

**Example:**

```
#include <machine.h>    /* Must include <machine.h>      */
char a;
void f(void)
{
    tas(&a);             /* Sets the result of a - 0      */
                        /* in CCR and performs          */
                        /* a |= 0x80                    */
}
```

```

void eepmov(void *dst,const void *src,unsigned char size)
void eepmov(void *dst,const void *src,unsigned int size)
void eepmovb(void *dst,const void *src,unsigned char size)
void eepmovw(void *dst,const void *src,unsigned int size)

```

**Description:** Transfers the bytes whose number is specified by the **size** from the address specified by **src** to the address specified by **dst** using the block transfer instruction, EEPMOV.

For the **eepmov** intrinsic function, **size** must be a constant value. The maximum size that can be specified is 255 in the **300** CPU/operating mode and 65535 in other modes. However, when the **size** is in the range of 256 to 65535, this function is expanded to EEPMOV.W. If interrupts are requested, do not use this function. If **size** is zero, no transfer occurs.

For the **eepmovb** and **eepmovw** intrinsic functions, size can be a variable. The **eepmovb** intrinsic function is always expanded to EEPMOV.B and the **eepmovw** intrinsic function to EEPMOV.W.

**Header:** <machine.h>

**Parameters:**

dst	Pointer to the destination
src	Pointer to the source
size	Transfer size

**Example:**

```

#include <machine.h> /* Must include <machine.h> */
char a[10],b[10];
void f(void)
{
    eepmov(b,a,10); /* The data of array a is          */
                   /* transferred to array b          */
                   /* using the EEPMOV instruction     */
}

```

**Remarks:** The **eepmovb** and **eepmovw** intrinsic functions are valid only when the CPU is H8SX and H8S(without legacy=v4 option).

**void eepmovi(void \*dst,const void \*src,unsigned int size)**

**Description:** Transfers the bytes whose number is specified by the **size** from the address specified by **src** to the address specified by **dst** using the block transfer instruction, EEPMOV. This function is expanded so that the EEPMOV instruction can resume transfer after returning from an interrupt.

**size** can be a constant or a variable. As a constant, up to 65535 can be specified. If **size** is zero, no transfer occurs. If **size** is a constant of less than 256 this function is expanded to one EEPMOV.B instruction. If **size** is a constant in the range from 256 to 510, this function is expanded to two EEPMOV.B instructions. If **size** is a constant no less than 512 or a variable, this function is expanded using EEPMOV.W as follows so that EEPMOV.W can resume transfer after an interrupt .

```
L1: EEPMOV.W
      MOV.W   R4,R4
      BNE     L1
```

**Header:** <machine.h>

**Parameters:**

dst	Pointer to the destination
src	Pointer to the source
size	Transfer size

**Example:**

```
#include <machine.h> /* Must include <machine.h> */
char a[10],b[10];
void f(void)
{
    eepmovi(b,a,10); /* The data of array a is */
                    /* transferred to array b */
                    /* using the EEPMOV instruction */
}
```

**Remarks:** This intrinsic function is valid only when the CPU is H8SX or H8S.



```

void movmdb(void *dst, const void *src, unsigned int count)
void movmdw(int *dst, const int *src, unsigned int count)
void movmdl(long *dst, const long *src, unsigned int count)

```

**Description:** The MOVMD.B, MOVMD.W or MOVMD.L instruction transfers a memory block of 1, 2 or 4 bytes, respectively, the number of times specified by **count** from the address specified by **src** to the address specified by **dst**. **count** takes the value from zero to 65535. If **count** is zero, however, it is interpreted as 65536.

**Header:** <machine.h>

**Parameters:**

src	Pointer to the source
dst	Pointer to the destination
size	Transfer count

**Example:**

```

#include <machine.h>      /* Must include <machine.h> */
char s1[100], d1[100];
int  s2[50],  d2[50]
long s4[25],  d4[25]
void f(void)
{
    movmdb(d1, s1, 100); /* MOVMD.B transfers 100 bytes */
                          /* from array s1 to array d1 */
    movmdw(d2, s2, 50);  /* MOVMD.W transfers 100 bytes */
                          /* from array s2 to array d2 */
    movmdl(d4, s4, 25);  /* MOVMD.L transfers 100 bytes */
                          /* from array s4 to array d4 */
}

```

**Remarks:** This intrinsic function is valid only when the CPU is H8SX.

## unsigned int movsd(char \*dst, const char \*src, unsigned int size)

**Description:** Transfers a memory block using the block transfer instruction MOVSD from the address specified by **src** to the address specified by **dst** either until a byte whose value is zero (H'00) has been transferred or until the transferred size has reached **size**. The return value is the value subtracting the size of actually-transferred bytes from the size given by **size**.  
**size** takes the value from zero to 65535. If **size** is zero, however, the maximum size allowed to transfer is interpreted as 65536.

**Header:** <machine.h>

**Return value:** The value subtracting the size actually transferred from the given **size**

**Parameters:**

src	Pointer to the source
dst	Pointer to the destination
size	Maximum size allowed to transfer

**Example:**

```
#include <machine.h>          /* Must include <machine.h> */
const char *s;
char d[100];
unsigned int remain;

void f(void)
{
    remain = movsd(d, s, 100); /* The string s is copied */
                               /* to the array d usingthe */
                               /* MOVSD instruction within*/
}                               /* the limit of 100 bytes */
```

**Remarks:** This intrinsic function is valid only when the CPU is H8SX.

## void nop(void)

**Description:** Expanded into a NOP instruction

**Header file:** <machine.h>

**Example:**

```
#include <machine.h>          /* Must include <machine.h> */
int a;
void f(void)
{
    while(a)nop();             /* Executes a NOP instruction */
                               /* while a!=0 */
}
```

```

int ovfaddc(char dst,char src,char *rst)
int ovfaddw(int dst, int src,int *rst)
int ovfaddl(long dst,long src,long *rst)
int ovfadduc(unsigned char dst,unsigned char src,unsigned char *rst)
int ovfadduw(unsigned int dst,unsigned int src,unsigned int *rst)
int ovfaddul(unsigned long dst,unsigned long src,unsigned long *rst)

```

**Description:** The functions **ovfaddc**, **ovfaddw**, and **ovfaddl** add signed 1-byte, 2-byte, and 4-byte data **dst** and **src**, respectively. The functions **ovfadduc**, **ovfadduw**, and **ovfaddul** add unsigned 1-byte, 2-byte, and 4-byte data **dst** and **src**, respectively. Then, these functions store the results to the area specified by **rst** only when **rst** is not 0, and return 0 when the results do not overflow or return a value other than 0 when they do overflow. These functions can be used only in the conditional statements such as **if**, **do**, **while**, and **for** statements. The **ovfaddl** and **ovfaddul** functions are valid when the CPU is other than **H8/300**.

**Header file:** <machine.h>

<b>Return value:</b>	When the result overflows	A value other than 0
	When the results does not overflow	0

<b>Parameters:</b>	dst, src	Operands of addition
	rst	Result storage area (The result is not stored if the rst value is 0)

**Example:**

```

#include <machine.h>    /* Must include <machine.h>    */
int dst, src;
void f(void)
{
    if(ovfaddw(dst,src,0) /* Determine the result of    */
        /* dst + src by BVC                            */
        dst=0;
}

```

```

int ovfsubc(char dst,char src,char *rst)
int ovfsubw(int dst,int src,int *rst)
int ovfsubl(long dst,long src,long *rst)
int ovfsubuc(unsigned char dst,unsigned char src,unsigned char *rst)
int ovfsubuw(unsigned int dst,unsigned int src,unsigned int *rst)
int ovfsubul(unsigned long dst,unsigned long src,unsigned long *rst)

```

Description: The functions **ovfsubc**, **ovfsubw**, and **ovfsubl** subtract signed 1-byte, 2-byte, and 4-byte data **dst** and **src**, respectively (dst-src). The functions **ovfsubuc**, **ovfsubuw**, and **ovfsubul** subtract unsigned 1-byte, 2-byte, and 4-byte data **dst** and **src**, respectively.

Then, these functions store the results to the area specified by **rst** only when **rst** is not 0, and return 0 when the results do not overflow or return a value other than 0 when they do overflow.

These functions can be used only in the conditional statements such as **if**, **do**, **while**, and **for** statements.

The **ovfsubl** and **ovfsubul** functions are valid when the CPU is other than **H8/300**.

Header file: <machine.h>

Return value:	When the result overflows	A value other than 0
	When the results does not overflow	0

Parameters:	dst, src	Operands of subtraction
	rst	Result storage area (The result is not stored if rst value is 0)

```

Example: #include <machine.h>    /* Must include <machine.h>    */
int dst,src;
void f(void)
{
    if(ovfsubw(dst,src,0) /* Determines the result of          */
                          /* dst - src by BVC                    */
        dst=0;
}

```

**int ovfshalc(char dst,char \*rst)**

**int ovfshalw(int dst,int \*rst)**

**int ovfshall(long dst,long \*rst)**

**Description:** The functions **ovfshalc**, **ovfshalw**, and **ovfshall** arithmetically shift 1-byte, 2-byte, and 4-byte data **dst** to the left by one bit, respectively, store the results to the area specified by **rst** only when **rst** is not 0, and return 0 when the results do not overflow or a value other than 0 when they do overflow. These functions can be used only in the conditional statements such as **if**, **do**, **while**, and **for** statements. The **ovfshalw** and **ovfshall** functions are valid when the CPU is other than **H8/300**.

**Header:** <machine.h>

<b>Return value:</b>	When the result overflows	A value other than 0
	When the results does not overflow	0

**Parameters:**

dst	Operand of bit shift operation
rst	Result storage area (The result is not stored if the rst value is 0)

**Example:**

```
#include <machine.h>    /* Must include <machine.h>    */
int dst;
void f(void)
{
    if(ovfshalw(dst,0))  /* Determines the result of    */
                        /* dst<<1 by BVC                */
        dst=0;
}
```

**int ovfshlluc(unsigned char dst,unsigned char \*rst)**

**int ovfshlluw(unsigned int dst,unsigned int \*rst)**

**int ovfshllul(unsigned long dst,unsigned long \*rst)**

**Description:** The functions **ovfshlluc**, **ovfshlluw**, and **ovfshllul** logically shift 1-byte, 2-byte, and 4-byte data **dst** to the left by one bit, respectively, store the results to the area specified by **rst** only when **rst** is not 0, and return 0 when the results do not overflow or a value other than 0 when they do overflow. These functions can be used only in the conditional statements such as **if**, **do**, **while**, and **for** statements. The **ovfshlluw** and **ovfshllul** functions are valid when the CPU is other than **H8/300**.

**Header:** <machine.h>

<b>Return value:</b>	When the result overflows	A value other than 0
	When the results does not overflow	0

**Parameters:**

<b>dst</b>	Operand of bit shift operation
<b>rst</b>	Result storage area (The result is not stored if rst value is 0)

**Example:**

```
#include <machine.h>    /* Must include <machine.h>    */
int dst;
void f(void)
{
    if(ovfshlluw(dst,0)) /* Determines the result of    */
                        /* dst<<1 by BCC                */
        dst=0;
}
```

**int ovfnegc(char dst,char \*rst)**

**int ovfnegw(int dst,int \*rst)**

**int ovfnegl(long dst,long \*rst)**

**Description:** The functions **ovfnegc**, **ovfnegw**, and **ovfnegl** calculate 2's complements of 1-byte, 2-byte, and 4-byte data **dst**, respectively, store the results to the area specified by **rst** only when **rst** is not 0, and return 0 when the results do not overflow or a value other than 0 when they do overflow.

These functions can be used only in the conditional statements such as **if**, **do**, **while**, and **for** statements.

The **ovfnegw** and **ovfnegl** functions are valid when the CPU is other than **H8/300**.

**Header:** <machine.h>

<b>Return value:</b>	When the result overflows	A value other than 0
	When the results does not overflow	0

**Parameters:**

<b>dst</b>	Operand of 2's complement calculation
<b>rst</b>	Result storage area (The result is not stored if rst value is 0)

**Example:**

```
#include <machine.h>      /* Must include <machine.h> */
int dst,rst;
void f(void)
{
    if(ovfnegw(dst,&rst)) /* Sets the result of dst in */
                          /* rst and branches depending */
                          /* on the borrow of the */
    dst=0;                /* result of -dst */
}
```

**void dadd(unsigned char size,char \*ptr1,char \*ptr2,char \*rst)**

**Description:** Adds **size**-byte data stored in the area starting from **ptr1** to **size**-byte data stored in the area starting from **ptr2** in decimal and stores the result to the **size**-byte area starting from **rst**. The **size** must be a constant from 1 to 255.

**Header:** <machine.h>

<b>Parameters:</b>	size	Data size
	ptr1, ptr2	Operands of addition in decimal
	rst	Result storage area

**Example:**

```
#include <machine.h>          /* Must include <machine.h> */
char ptr1[5]={0x01,0x23,0x45,0x67,0x89}; /* 12345678910 */
char ptr2[5]={0x01,0x23,0x45,0x67,0x89}; /* 12345678910 */
char rst[5];
void main(void)
{
    dadd((char)5,ptr1,ptr2,rst);
                                /* Adds ptr1 and ptr2 for a      */
                                /* 10-digit decimal              */
}                                /* rst=0x02,0x46,0x91,0x35,0x78 */
```



**void dsub(unsigned char size,char \*ptr1,char \*ptr2,char \*rst)**

**Description:** Subtracts **size**-byte data stored in the area starting from **ptr2** from **size**-byte data stored in the area starting from **ptr1** in decimal and stores the result to the **size**-byte area starting from **rst**. The **size** must be a constant from 1 to 255.

**Header:** <machine.h>

<b>Parameters:</b>	size	Data size
	ptr1, ptr2	Operands of subtraction in decimal
	rst	Result storage area

**Example:**

```
#include <machine.h>          /* Must include <machine.h> */
char ptr1[5]={0x10,0x00,0x00,0x00,0x00}; /* 100000000010 */
char ptr2[5]={0x01,0x23,0x45,0x67,0x89}; /* 012345678910 */
char rst[5];
void main(void)
{
    dsub((char)5,ptr1,ptr2,rst);
                                /* Subtracts ptr2 from ptr1      */
                                /* for a 10-digit decimal          */
                                /* rst=0x08,0x76,0x54,0x32,0x11    */
}
```

## 10.3 C/C++ Libraries

### 10.3.1 Standard C Libraries

#### Overview of Libraries

This section describes the specifications of the C library functions, which can be used generally in C/C++ programs.. This section gives an overview of the library configuration, and describes the layout and the terms used in this library function description. Then, the specifications of each library is described according to the configuration of the library.

#### (1) Library Types

A library implements standard processing such as input/output and string manipulation in the form of C/C++ language functions. Libraries can be used by including standard include files for each unit of processing.

Standard include files contain declarations for the corresponding libraries and definitions of the macro names necessary to use them.

Table 10.29 shows the various library types and the corresponding standard include files.

**Table 10.29 Library Types and Corresponding Standard Include Files**

<b>Library Type</b>	<b>Description</b>	<b>Standard Include Files</b>
Program diagnostics	Outputs program diagnostic information.	<assert.h>
Character handling	Handles and checks characters.	<ctype.h>
Mathematics	Performs numerical calculations such as trigonometric functions.	<math.h> <mathf.h>
Non-local jumps	Supports transfer of control between functions.	<setjmp.h>
Variable arguments	Supports access to variable arguments for functions with such arguments.	<stdarg.h>
Input/output	Performs input/output handling. By using <no_float.h>, I/O functions that do not support floating-point numbers can be provided.	<stdio.h> <no_float.h>
General utilities	Performs C program standard processing such as storage area management.	<stdlib.h>
String handling	Performs string comparison, copying, etc.	<string.h>

In addition to the above standard include files, standard include files consisting solely of macro name definitions, shown in table 10.30, are provided to improve programming efficiency.

**Table 10.30 Standard Include Files Comprising Macro Name Definitions**

<b>Standard Include File</b>	<b>Description</b>
<stddef.h>	Defines macro names used by the standard include files.
<float.h>	Defines various limit values relating to the internal representation of floating-point numbers.
<limits.h>	Defines various limit values relating to compiler internal processing.
<errno.h>	Defines the value to set in errno when an error is generated in a library function.

## (2) Organization of Library Part

The organization of the library part of this manual is described below.

Library functions are categorized for each standard include file, and descriptions are given for each standard include file. For each category, there is first a description relating to the macro names and function declarations defined in the standard include file (figure 10.3), followed by a description of each function (figure 10.4).

Figure 10.3 shows the standard include file description layout, and figure 10.4, the function description layout.

<standard include file name>

- Summarizes the overall function of this standard include file.
- Describes names defined or declared in this standard include file according to the name categories such as [Type], [Constant], [Variable], and [Function]. For macro names, (macro) is always attached beside the name category or name description.
- Adds description if implementation-defined specifications are included or notes common to the functions declared in this standard include file are given.

**Figure 10.3 Layout of Standard Include File Description**

## Function name (return value and parameter names)

Description: Describes the library function.

Header file: Shows the name of standard include file to be declared.

Return value: Normal: Shows the return value when the library function ends normally.

Abnormal: Shows the return value when the library function ends abnormally.

Parameters: Indicates the meanings of the parameters.

Example: Describes the calling procedure.

Error conditions:

Conditions for the occurrence of errors that cannot be determined from the return value in library function processing.

If such an error occurs, the value defined in each compiler for the error type is set in `errno`.\*.

Remarks: Details the library function specifications or notes on use.

Implementation define:

The compiler processing method.

**Figure 10.4 Layout of Function Description**

Note: \* **errno** is a variable that stores the error type if an error occurs during execution of a library function. See section 10.3.1, descriptions for `<stddef.h>`, for details.

### (3) Terms Used in Library Function Descriptions

#### (a) Stream input/output

In data input/output, it would lead to poor efficiency if each call of an input/output function handling a single character drove the input/output device and 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.

## (b) 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. When a file is opened, the file pointer value must be checked to see whether the open processing has been successful.

## (c) 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:

- (i) Macros are expanded automatically by the preprocessor, and therefore a macro cannot be invalidated even if the user declares a function with the same name.
- (ii) If an expression with a side effect is specified as a macro parameter (assignment expression, increment, decrement), the 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.

## (d) EOF

In functions such as **getc**, **getchar**, and **fgetc**, which input data from a file, EOF is the value returned at end-of-file. The name EOF is defined in the `<stdio.h>` standard include file.

(e) NULL

This is the value when a pointer is not pointing at anything. The name NULL is defined in the `<stddef.h>` standard include file.

(f) Null characters

The end of a string literal 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.

(g) 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 as the return code.

(h) 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.

(i) 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 designator (`\n`) is input as a line separator. In output, output of the current line is terminated by outputting the new-line designator (`\n`). Text files are used to input/output files that store standard text for each implementation. With text files, characters input or output by a library function do not necessarily correspond to a physical arrangement of data in the file.

(ii) Binary files

A binary file is configured as a row of byte data. Data input or output by a library function correspond to a physical list of data in the file.

(i) Standard input/output files

Files that can be used as standard by input/output library functions without preparations such as file opening 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`).

(i) Standard input file (`stdin`)

Standard file comprising input to a program.

(ii) Standard output file (`stdout`)

Standard file comprising output from a program.

(iii) Standard error output file (`stderr`)

Standard file for performing output of error messages, etc., from a program.

(j) Floating-point numbers

Floating-point numbers are numbers represented by approximation of real-numbers. In a C/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:

$$2^n \times m \text{ (n: integer, m: binary fraction)}$$

Here, n is called the exponent of the floating-point number, and m is called the mantissa.

The number of bits to represent n and m is 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.

(i) 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.

(ii) Rounding

Rounding is performed when an intermediate result of an operation of higher precision than a floating-point number is stored as a floating-point number. There is rounding up, rounding down, and half-adjust rounding (rounding up fractions over 1/2 and rounding down fractions under 1/2; or, in binary representation, rounding down 0 and rounding up 1).

(iii) Normalization

When a floating-point number is represented in the form  $2^n \times m$ , the same number can be represented in different ways.

Example: The following two expressions represent the same value.

$$2^5 \times 1.0_{(2)} \quad (_{(2)} \text{ indicates a binary number})$$

$$2^6 \times 0.1_{(2)}$$

Usually, a representation in which the leading digit is not 0 like the former expression 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.

(iv) Guard bit

When saving an intermediate result of a floating-point operation, data one bit longer than the actual floating-point number is normally provided in order to carry out rounding. However, this alone does not permit an accurate result to be achieved in the event of cancellation of significant digits, etc. For this reason, the intermediate result is saved with an extra bit, called a guard bit.

(k) 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 strings, as shown in table 10.31.

**Table 10.31 File Access Modes**

<b>Access Mode</b>	<b>Meaning</b>
'r'	Open text file for reading
'w'	Open text file for writing
'a'	Open text file for addition
'rb'	Open binary file for reading
'wb'	Open binary file for writing
'ab'	Open binary file for addition
'r+'	Open text file for reading and updating
'w+'	Open text file for writing and updating
'a+'	Open text file for addition and updating
'r+b'	Open binary file for reading and updating
'w+b'	Open binary file for writing and updating
'a+b'	Open binary file for addition and updating

(l) Implementation definition

Definitions differ depending on compilers.

(m) 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.

(n) 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.



#### (4) Notes on use of libraries

- (a) The contents of macros defined in a library differ in each compiler. When a library is used, the behavior is not guaranteed if the contents of these macros are redefined.
- (b) 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.

#### **<stddef.h>**

Defines macro names used in common in the standard include file.

The following macro names are all implementation-defined.

Type	Definition Name	Description
Type (macro)	ptrdiff_t	Indicates the type of the result of subtracting two pointers.
	size_t	Indicates the type of the result of the sizeof operator.
Constant (macro)	NULL	Indicates the value when a pointer is not pointing at anything. This value is such that the result of a comparison with 0 using the equality operator (==) is true.
Variable (macro)	errno	If an error occurs during library function processing, the error code defined in the respective library is set in errno. By setting 0 in errno before calling a library function and checking the error code set in errno after the library function processing has ended, it is possible to check whether an error occurred during the library function processing.
Function(macro)	offsetof (type, member)	Obtains the offset in bytes from the beginning of a structure to a structure member.

## Implementation Define

Definition Name	Description	
Value of macro NULL	The pointer type value 0 is set to void.	
Contents of macro ptrdiff_t	int type	H8SX normal mode, H8SX middle mode, H8SX advanced mode with ptr16 option, H8SX maximum mode with ptr16 option, H8S/2600 normal mode, H8S/2000 normal mode, H8S/2600 advanced mode with ptr16 option, H8S/2000 advanced mode with ptr16 option, H8/300H normal mode, H8/300
	long type	H8SX advanced mode without ptr16 option, H8SX maximum mode without ptr16 option, H8S/2600 advanced mode without ptr16 option, H8S/2000 advanced mode without ptr16 option, H8/300H advanced mode

## <assert.h>

Adds diagnostics into programs.

Type	Definition Name	Description
Function (macro)	<code>assert</code>	Adds diagnostics into programs.

To invalidate the diagnostics defined by <assert.h>, define macro name **NDEBUG** with a **#define** statement (**#define NDEBUG**) before including <assert.h>.

Note: If an **#undef** statement is used for macro name **assert**, the result of subsequent **assert** calls is not guaranteed.

### **void assert(int expression)**

Description: Adds diagnostics into programs.

Header file: <assert.h>

Parameters: expression Expression to be evaluated.

Example:

```
#include <assert.h>
int expression;
assert (expression);
```

Remarks: When **expression** is true, the **assert** macro terminates processing without returning a value. If **expression** is false, it outputs diagnostic information to the standard error file in the form defined by the compiler, and then calls the **abort** function.

The diagnostic information includes the parameter program text, source file name, and source line numbers.

Implementation define:

The following message is output when the expression is false for **assert (expression)**:

ASSERTION FAILED:ΔexpressionΔFILEΔ<file name>,lineΔ<line number>

## <ctype.h>

Performs type determination and conversion for characters.

Type	Definition Name	Description
Function	isalnum	Tests for an alphabetic character or a decimal digit.
	isalpha	Tests for an alphabetic character.
	isctrl	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.

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 10.32.

**Table 10.32 Character Types**

<b>Character Type</b>	<b>Description</b>
Uppercase letter	Any of the following 26 characters 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
Lowercase letter	Any of the following 26 characters 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'
Alphabetic character	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 ( ' '), an alphabetic character, or a decimal digit

## Implementation Define

<b>Item</b>	<b>Compiler Specifications</b>
Character set inspected by the <code>isalnum</code> , <code>isalpha</code> , <code>iscntrl</code> , <code>islower</code> , <code>isprint</code> , and <code>isupper</code> functions	Character set represented by the unsigned char type. Table 10.33 shows the character set that results in a true return value.

**Table 10.33 True Characters**

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'

**int isalnum(int c)**

Description: Tests for an alphabetic character or decimal digit.

Header file: <ctype.h>

Return values: If character **c** is an alphabetic character or a decimal digit: Nonzero  
 If character **c** is not an alphabetic character or a decimal digit: 0

Parameters: **c** Character to be tested

Example:

```
#include <ctype.h>
int c, ret;
ret=isalnum(c);
```

## **int isalpha(int c)**

Description: Tests for an alphabetic character.

Header file: <ctype.h>

Return values: If character **c** is an alphabetic character : Nonzero  
If character **c** is not an alphabetic character : 0

Parameters: **c** Character to be tested

Example:

```
#include <ctype.h>
int c, ret;
ret=isalpha(c);
```

## **int iscntrl(int c)**

Description: Tests for a control character.

Header file: <ctype.h>

Return values: If character **c** is a control character: Nonzero  
If character **c** is not a control character: 0

Parameters: **c** Character to be tested

Example:

```
#include <ctype.h>
int c, ret;
ret=iscntrl (c);
```

## **int isdigit(int c)**

Description: Tests for a decimal digit.

Header file: <ctype.h>

Return values: If character **c** is a decimal digit: Nonzero  
If character **c** is not a decimal digit: 0

Parameters: **c** Character to be tested

Example:

```
#include <ctype.h>
int c, ret;
ret=isdigit(c);
```

## **int isgraph(int c)**

Description: Tests for any printing character except space (' ').

Header file: <ctype.h>

Return values: If character **c** is a printing character except space: Nonzero  
If character **c** is not a printing character except space: 0

Parameters: **c** Character to be tested

Example:

```
#include <ctype.h>
int c, ret;
ret=isgraph(c);
```



## **int islower(int c)**

Description: Tests for a lowercase letter.

Header file: <ctype.h>

Return values: If character **c** is a lowercase letter: Nonzero  
If character **c** is not a lowercase letter: 0

Parameters: **c** Character to be tested

Example:

```
#include <ctype.h>
int c, ret;
ret=islower(c);
```

## **int isprint(int c)**

Description: Tests for a printing character, including space (' ').

Header file: <ctype.h>

Return values: If character **c** is a printing character, including space: Nonzero  
If character **c** is not a printing character, including space: 0

Parameters: **c** Character to be tested

Example:

```
#include <ctype.h>
int c, ret;
ret=isprint(c);
```

## **int ispunct(int c)**

Description: Tests for a special character.

Header file: <ctype.h>

Return values: If character **c** is a special character: Nonzero  
If character **c** is not a special character: 0

Parameters: **c** Character to be tested

Example: 

```
#include <ctype.h>
int c, ret;
ret=ispunct(c);
```

## **int isspace(int c)**

Description: Tests for a white-space character.

Header file: <ctype.h>

Return values: If character **c** is a white-space character: Nonzero  
If character **c** is not a white-space character: 0

Parameters: **c** Character to be tested

Example: 

```
#include <ctype.h>
int c, ret;
ret=isspace(c);
```

## **int isupper(int c)**

Description: Tests for an uppercase letter.

Header file: <ctype.h>

Return values: If character **c** is an uppercase letter: Nonzero  
If character **c** is not an uppercase letter: 0

Parameters: **c** Character to be tested

Example: 

```
#include <ctype.h>
int c, ret;
ret=isupper(c);
```

## **int isxdigit(int c)**

Description: Tests for a hexadecimal digit.

Header file: <ctype.h>

Return values: If character **c** is a hexadecimal digit: Nonzero  
If character **c** is not a hexadecimal digit: 0

Parameters: **c** Character to be tested

Example: 

```
#include <ctype.h>
int c, ret;
ret=isxdigit(c);
```

## **int tolower(int c)**

Description: Converts an uppercase letter to the corresponding lowercase letter.

Header file: <ctype.h>

Return values: If character **c** is an uppercase letter: Lowercase letter corresponding to character **c**  
If character **c** is not an uppercase letter: Character **c**

Parameters: **c** Character to be converted

Example: 

```
#include <ctype.h>
int c, ret;
ret=tolower(c);
```

## **int toupper(int c)**

Description: Converts a lowercase letter to the corresponding uppercase letter.

Header file: <ctype.h>

Return values: If character **c** is a lowercase letter: Uppercase letter corresponding to character **c**  
If character **c** is not a lowercase letter: Character **c**

Parameters: **c** Character to be converted

Example: 

```
#include <ctype.h>
int c, ret;
ret=toupper(c);
```

## <float.h>

Defines various limits relating to the internal representation of floating-point numbers.

The following macro names are all implementation-defined.

Type	Definition Name	Definition Value	Description
Constant (macro)	FLT_RADIX	2	Indicates the radix in exponent representation.
	FLT_ROUNDS	1	Indicates whether or not the result of an add operation is rounded off. The meaning of this macro definition is as follows:  (1) When result of add operation is rounded off: Positive value  (2) When result of add operation is rounded down: 0  (3) When nothing is specified: -1 The rounding-off and rounding-down methods are implementation-defined.
	FLT_GUARD	1	Indicates whether or not a guard bit is used in multiply operations. The meaning of this macro definition is as follows:  (1) When guard bit is used: 1  (2) When guard bit is not used: 0
	FLT_NORMALIZE	1	Indicates whether or not floating-point values are normalized. The meaning of this macro definition is as follows:  (1) When normalized: 1  (2) When not normalized: 0
	FLT_MAX	3.4028235677973364e+38F	Indicates the maximum value that can be represented as a float type floating-point value.
	DBL_MAX	1.7976931348623158e+308	Indicates the maximum value that can be represented as a double type floating-point value.
	LDBL_MAX	1.7976931348623158e+308	Indicates the maximum value that can be represented as a long double type floating-point value.

Type	Definition Name	Definition Value	Description
Constant (macro)	FLT_MAX_EXP	127	Indicates the power-of-radix maximum value that can be represented as a float type floating-point value.
	DBL_MAX_EXP	1023	Indicates the power-of-radix maximum value that can be represented as a double type floating-point value.
	LDBL_MAX_EXP	1023	Indicates the power-of-radix maximum value that can be represented as a long double type floating-point value.
	FLT_MAX_10_EXP	38	Indicates the power-of-10 maximum value that can be represented as a float type floating-point value.
	DBL_MAX_10_EXP	308	Indicates the power-of-10 maximum value that can be represented as a double type floating-point value.
	LDBL_MAX_10_EXP	308	Indicates the power-of-10 maximum value that can be represented as a long double type floating-point value.
	FLT_MIN	1.175494351e-38F	Indicates the minimum positive value that can be represented as a float type floating-point value.
	DBL_MIN	2.2250738585072014e-308	Indicates the minimum positive value that can be represented as a double type floating-point value.
	LDBL_MIN	2.2250738585072014e-308	Indicates the minimum positive value that can be represented as a long double type floating-point value.
	FLT_MIN_EXP	-149	Indicates the power-of-radix minimum value of a floating-point value that can be represented as a float type positive value.
	DBL_MIN_EXP	-1074	Indicates the power-of-radix minimum value of a floating-point value that can be represented as a double type positive value.
	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.

Type	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 double type floating-point value decimal-precision.
	LDBL_DIG	15	Indicates the maximum number of digits in long double type floating-point value decimal-precision.
	FLT_MANT_DIG	24	Indicates the maximum number of mantissa digits when a float type floating-point value is represented in the radix.
	DBL_MANT_DIG	53	Indicates the maximum number of mantissa digits when a double type floating-point value is represented in the radix.
	LDBL_MANT_DIG	53	Indicates the maximum number of mantissa digits when a long double type floating-point value is represented in the radix.
	FLT_EXP_DIG	8	Indicates the maximum number of exponent digits when a float type floating-point value is represented in the radix.
	DBL_EXP_DIG	11	Indicates the maximum number of exponent digits when a double type floating-point value is represented in the radix.
	LDBL_EXP_DIG	11	Indicates the maximum number of exponent digits when a long double type floating-point value is represented in the radix.

Type	Definition Name	Definition Value	Description
Constant (macro)	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.
	FLT_POS_EPS_EXP	-23	Indicates the minimum integer n for which $1.0 + (\text{radix})^n \neq 1.0$ in float type.
	DBL_POS_EPS_EXP	-52	Indicates the minimum integer n for which $1.0 + (\text{radix})^n \neq 1.0$ in double type.
	LDBL_POS_EPS_EXP	-52	Indicates the minimum integer n for which $1.0 + (\text{radix})^n \neq 1.0$ in long double type.
	FLT_NEG_EPS_EXP	-24	Indicates the minimum integer n for which $1.0 - (\text{radix})^n \neq 1.0$ in float type.
	DBL_NEG_EPS_EXP	-53	Indicates the minimum integer n for which $1.0 - (\text{radix})^n \neq 1.0$ in double type.
	LDBL_NEG_EPS_EXP	-53	Indicates the minimum integer n for which $1.0 - (\text{radix})^n \neq 1.0$ in long double type.



## <limits.h>

Defines various limits relating to the internal representation of integer type data.

The following macro names are all implementation-defined.

Type	Definition Name	Definition Value	Description
Constant (macro)	CHAR_BIT	8	Indicates the number of bits of which char type is composed.
	CHAR_MAX	127	Indicates the maximum value that a char type variable can have as a value.
	CHAR_MIN	-128	Indicates the minimum value that a char type variable can have as a value.
	SCHAR_MAX	127	Indicates the maximum value that a signed char type variable can have as a value.
	SCHAR_MIN	-128	Indicates the minimum value that a signed char type variable can have as a value.
	UCHAR_MAX	255u	Indicates the maximum value that an unsigned char type variable can have as a value.
	SHRT_MAX	32767	Indicates the maximum value that a short type variable can have as a value.
	SHRT_MIN	-32768	Indicates the minimum value that a short type variable can have as a value.
	USHRT_MAX	65535u	Indicates the maximum value that an unsigned short type variable can have as a value.
	INT_MAX	32767	Indicates the maximum value that an int type variable can have as a value.
	INT_MIN	-32768	Indicates the minimum value that an int type variable can have as a value.
	UINT_MAX	65535u	Indicates the maximum value that an unsigned int type variable can have as a value.
	LONG_MAX	2147483647	Indicates the maximum value that a long type variable can have as a value.
	LONG_MIN	-2147483647L-1L	Indicates the minimum value that a long type variable can have as a value.
	ULONG_MAX	4294967295u	Indicates the maximum value that an unsigned long type variable can have as a value.

## <errno.h>

Defines the value to set in **errno** when an error is generated in a library function.

The following macro names are all implementation-defined.

Type	Definition Name	Description
Variable (macro)	errno	int type variable. An error number is set when an error is generated in a library function.
Constant (macro)	ERANGE	Refer to section 12.3, C Library Error Messages.
	EDOM	Same as above
	EDIV	Same as above
	ESTRN	Same as above
	PTRERR	Same as above
	ECBASE	Same as above
	ETLN	Same as above
	EEXP	Same as above
	EEXPN	Same as above
	EFLOATO	Same as above
	EFLOATU	Same as above
	EDBLO	Same as above
	EDBLU	Same as above
	ELDBLO	Same as above
	ELDBLU	Same as above
	NOTOPN	Same as above
	EBADF	Same as above
	ECSPEC	Same as above

## <math.h>

Performs various mathematical operations.

The following macro names are all implementation-defined.

Type	Definition Name	Description
Constant (macro)	EDOM	Indicates the value to be set in errno if the value of an parameter input to a function is outside the range of values defined in the function.
	ERANGE	Indicates the value to be set in errno if the result of a function cannot be represented as a double type value, or if overflow or underflow occurs.
	HUGE_VAL	Indicates the value for the function return value if the result of a function overflows.
Function	acos	Computes the arc cosine of a floating-point number.
	asin	Computes the arc sine of a floating-point number.
	atan	Computes the arc tangent of a floating-point number.
	atan2	Computes the arc tangent of the result of a division of two floating-point numbers.
	cos	Computes the cosine of a floating-point radian value.
	sin	Computes the sine of a floating-point radian value.
	tan	Computes the tangent of a floating-point radian value.
	cosh	Computes the hyperbolic cosine of a floating-point number.
	sinh	Computes the hyperbolic sine of a floating-point number.
	tanh	Computes the hyperbolic tangent of a floating-point number.
	exp	Computes the exponential function of a floating-point number.
	frexp	Breaks a floating-point number into a [0.5, 1.0] value and a power of 2.
	ldexp	Multiplies a floating-point number by a power of 2.
	log	Computes the natural logarithm of a floating-point number.
	log10	Computes the base-ten logarithm of a floating-point number.
	modf	Breaks a floating-point number into integral and fractional parts.
	pow	Computes a power of a floating-point number.
	sqrt	Computes the positive square root of a floating-point number.
	ceil	Returns the smallest integral value not less than the given floating-point number.
	fabs	Computes the absolute value of a floating-point number.
	floor	Returns the largest integral value not greater than the given floating-point number.
	fmod	Computes the remainder of 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 depends on the implementation.

#### (2) Range error

A range error occurs if the result of a function cannot be represented as a **double** type value. In this case, the value of **ERANGE** is set in **errno**. If the result overflows, the function returns the value of **HUGE\_VAL**, with the same sign as the correct value of the function. If the result underflows, 0 is returned as the return value.

### Notes

- (1) If there is a possibility of a domain error resulting from a `<math.h>` function call, it is dangerous to use the resultant value directly. The value of **errno** should always be checked before using the result in such cases.

#### Example:

```

    .
    .
    .
1   x=asin(a);
2   if (errno==EDOM)
3       printf ("error\n");
4   else
5       printf ("result is : %lf\n", x);
    .
    .
    .
```

In line 1, the arc sine value is computed using the **asin** function. If the value of parameter **a** is outside the domain of the **asin** function  $[-1.0, 1.0]$ , the **EDOM** value is set in **errno**. Line 2 determines whether a domain error has occurred. If a domain error has occurred, error is output in line 3. If there is no domain error, the arc sine value is output in line 5.

(2) Whether or not a range error occurs depends on the internal representation format of floating-point number determined by the compiler. For example, if an internal representation format that allows infinity to be represented as a value is used, `<math.h>` library functions can be implemented without causing range errors.

**Implementation Define**

Item	Compiler Specifications
Value returned by a mathematical function if an input parameter is out of the range	A not-a-number is returned. For details on the format of not-a-number, refer to section 10.1.3, Floating-Point Number Specifications.
Is <code>errno</code> set to the value of macro <code>ERANGE</code> if an underflow error occurs in a mathematical function?	Not specified.
Does a range error occur if the second argument in the <code>fmod</code> function is 0?	A range error occurs.

**double acos(double d)**

Description:      Computes the arc cosine of a floating-point number.

Header file:      `<math.h>`

Return values:    Normal:      Arc cosine of **d**  
                  Abnormal:    In case of domain error: Returns not-a-number.

Parameters:      **d**              Floating-point number for which arc cosine is to be computed

Example:          `#include <math.h>`  
                  `double d, ret;`  
                  `ret=acos(d);`

Error conditions:              A domain error occurs for a value of **d** not in the range  $[-1.0, 1.0]$ .

Remarks:          The **acos** function returns the arc cosine in the range  $[0, \pi]$  by the radian.

## **double asin (double d)**

Description: Computes the arc sine of a floating-point number.

Header file: <math.h>

Return values: Normal: Arc sine of **d**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: **d** Floating-point number for which arc sine is to be computed

Example:

```
#include <math.h>
double d, ret;
ret=asin(d);
```

Error conditions:

A domain error occurs for a value of **d** not in the range  $[-1.0, 1.0]$ .

Remarks: The **asin** function returns the arc sine in the range  $[-\pi/2, \pi/2]$  by the radian.

## **double atan(double d)**

Description: Computes the arc tangent of a floating-point number.

Header file: <math.h>

Return values: Arc tangent of **d**

Parameters: **d** Floating-point number for which arc tangent is to be computed

Example:

```
#include <math.h>
double d, ret;
ret=atan(d);
```

Remarks: The **atan** function returns the arc tangent in the range  $(-\pi/2, \pi/2)$  by the radian.

## double atan2(double y, double x)

Description: Computes the arc tangent of division of two floating-point numbers.

Header file: <math.h>

Return values: Normal: Arc tangent value when **y** is divided by **x**.  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: x Divisor  
y Dividend

Example: 

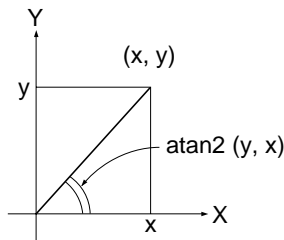
```
#include <math.h>
double x, y, ret;
ret=atan2(y, x);
```

Error conditions:

A domain error occurs if the values of both **x** and **y** are 0.0.

Remarks: The **atan2** function returns the arc tangent in the range  $(-\pi, \pi]$  by the radian. The meaning of the **atan2** function is illustrated in figure 10.5. 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  $\pi$ . If **x** = 0.0, the result is  $\pm\pi/2$ , depending on whether **y** is positive or negative.



**Figure 10.5 Meaning of atan2 Function**

## **double cos(double d)**

Description: Computes the cosine of a floating-point radian value.

Header file: <math.h>

Return values: Cosine of **d**

Parameters: **d** Radian value for which cosine is to be computed

Example:

```
#include <math.h>
double d, ret;
ret=cos(d);
```

## **double sin(double d)**

Description: Computes the sine of a floating-point radian value.

Header file: <math.h>

Return values: Sine of **d**

Parameters: **d** Radian value for which sine is to be computed

Example:

```
#include <math.h>
double d, ret;
ret=sin(d);
```



## **double tan(double d)**

Description: Computes the tangent of a floating-point radian value.

Header file: <math.h>

Return values: Tangent of **d**

Parameters: **d** Radian value for which tangent is to be computed

Example:

```
#include <math.h>
double d, ret;
ret=tan(d);
```

## **double cosh(double d)**

Description: Computes the hyperbolic cosine of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic cosine of **d**

Parameters: **d** Floating-point number for which hyperbolic cosine is to be computed

Example:

```
#include <math.h>
double d, ret;
ret=cosh(d);
```

## **double sinh(double d)**

Description: Computes the hyperbolic sine of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic sine of **d**

Parameters: **d** Floating-point number for which hyperbolic sine is to be computed

Example: 

```
#include <math.h>
double d, ret;
ret=sinh(d);
```

## **double tanh(double d)**

Description: Computes the hyperbolic tangent of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic tangent of **d**

Parameters: **d** Floating-point number for which hyperbolic tangent is to be computed

Example: 

```
#include <math.h>
double d, ret;
ret=tanh(d);
```

## double exp(double d)

Description: Computes the exponential function of a floating-point number.

Header file: <math.h>

Return values: Exponential value of **d**

Parameters: **d** Floating-point number for which exponential function is to be computed

Example:

```
#include <math.h>
double d, ret;
ret=exp(d);
```

## double frexp(double value, double ret)

Description: Breaks a floating-point number into a [0.5, 1.0] value and a power of 2.

Header file: <math.h>

Return values: If value is 0.0: 0.0  
If value is not 0.0: Value of **ret** defined by  $\text{ret} * 2^{(e)} = \text{value}$

Parameters: **value** Floating-point number to be broken into a [0.5, 1.0] value and a power of 2  
**e** Pointer to storage area that holds power-of-2 value

Example:

```
#include <math.h>
double ret, value;
int *e;
ret=frexp(value, e);
```

Remarks: The **frexp** function breaks a 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 **e**.

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 **e** and the value of **ret** are both 0.0.

## **double ldexp(double ret, int f)**

Description: Multiplies a floating-point number by a power of 2.

Header file: <math.h>

Return values: Result of  $e * 2^f$  operation

Parameters: **e** Floating-point number to be multiplied by a power of 2  
**f** Power-of-2 value

Example:

```
#include <math.h>
double ret, e;
int f;
ret=ldexp(e, f);
```

## **double log(double d)**

Description: Computes the natural logarithm of a floating-point number.

Header file: <math.h>

Return values: Normal: Natural logarithm of **d**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: **d** Floating-point number for which natural logarithm is to be computed

Example:

```
#include <math.h>
double d, ret;
ret=log(d);
```

Error conditions:

A domain error occurs if **d** is negative.

A range error occurs if **d** is 0.0.

## **double log10(double d)**

Description: Computes the base-ten logarithm of a floating-point number.

Header file: <math.h>

Return values: Normal: Base-ten logarithm of **d**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: **d** Floating-point number for which base-ten logarithm is to be computed

Example: 

```
#include <math.h>
double d, ret;
ret=log10(d);
```

Error conditions:

A domain error occurs if **d** is negative.

A range error occurs if **d** is 0.0.

## **double modf(double a, double\*b)**

Description: Breaks a floating-point number into integral and fractional parts.

Header file: <math.h>

Return values: Fractional part of **a**

Parameters: **a** Floating-point number to be broken into integral and fractional parts  
**b** Pointer indicating storage area that stores integral part

Example: 

```
#include <math.h>
double a, *b, ret;
ret=modf(a, b);
```

## **double pow(double x, double y)**

Description: Computes a power of floating-point number.

Header file: <math.h>

Return values: Normal: Value of **x** raised to the power **y**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: **x** Value to be raised to a power  
**y** Power value

Example: 

```
#include <math.h>
double x, y, ret;
ret=pow(x, y);
```

Error conditions:

A domain error occurs if **x** is 0.0 and **y** is 0.0 or less, or if **x** is negative and **y** is not an integer.

## **double sqrt(double d)**

Description: Computes the positive square root of a floating-point number.

Header file: <math.h>

Return values: Normal: Positive square root of **d**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: **d** Floating-point number for which positive square root is to be computed

Example: 

```
#include <math.h>
double d, ret;
ret=sqrt(d);
```

Error conditions:

A domain error occurs if **d** is negative.

## **double ceil(double d)**

Description: Returns the smallest integral value not less than the given floating-point number.

Header file: <math.h>

Return values: Smallest integral value not less than **d**

Parameters: **d** Floating-point number for which smallest integral value not less than that number is to be computed

Example: 

```
#include <math.h>
double d, ret;
ret=ceil(d);
```

Remarks: The **ceil** function returns the smallest integral value not less than **d**, expressed as a **double**. Therefore, if **d** is negative, the value after truncation of the fractional part is returned.

## **double fabs(double d)**

Description: Computes the absolute value of a floating-point number.

Header file: <math.h>

Return values: Absolute value of **d**

Parameters: **d** Floating-point number for which absolute value is to be computed

Example: 

```
#include <math.h>
double d, ret;
ret=fabs(d);
```

## **double floor(double d)**

Description: Returns the largest integral value not greater than the given floating-point number.

Header file: <math.h>

Return values: Largest integral value not greater than **d**

Parameters: **d** Floating-point number for which largest integral value not greater than that number is to be computed

Example:

```
#include <math.h>
double d, ret;
ret=floor(d);
```

Remarks: The **floor** function returns the largest integral value not greater than **d**, expressed as a **double**. Therefore, if **d** is negative, the value after rounding-up of the fractional part is returned.

## **double fmod(double x, double y)**

Description: Computes the remainder of division of two floating-point numbers.

Header file: <math.h>

Return values: When **y** is 0.0: **x**  
When **y** is not 0.0: Remainder of division of **x** by **y**

Parameters: **x** Dividend  
**y** Divisor

Example:

```
#include <math.h>
double x, y, ret;
ret=fmod(x, y);
```

Remarks: In the **fmod** function, the relationship between parameters **x** and **y** and return value **ret** is as follows:

$x = y * i + \text{ret}$  (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 expressed, the value of the result is undefined.



## <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 a **float**-type parameter and returns a **float**-type value.

The following constants (macros) are all implementation-defined.

Type	Definition Name	Description
Constant (macro)	EDOM	Indicates the value to be set in errno if the value of an parameter input to a function is outside the range of values defined in the function.
	ERANGE	Indicates the value to be set in errno if the result of a function cannot be represented as a float value, or if overflow or underflow occurs.
	HUGE_VAL	Indicates the value for the function return value if the result of a function overflows.
Function	acosf	Computes the arc cosine of a floating-point number.
	asinf	Computes the arc sine of a floating-point number.
	atanf	Computes the arc tangent of a floating-point number.
	atan2f	Computes the arc tangent of the result of a division of two floating-point numbers.
	cosf	Computes the cosine of a floating-point radian value.
	sinf	Computes the sine of a floating-point radian value.
	tanf	Computes the tangent of a floating-point radian value.
	coshf	Computes the hyperbolic cosine of a floating-point number.
	sinhf	Computes the hyperbolic sine of a floating-point number.
	tanhf	Computes the hyperbolic tangent of a floating-point number.
	expf	Computes 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	Computes the natural logarithm of a floating-point number.
	log10f	Computes the base-ten logarithm of a floating-point number.
	modff	Breaks a floating-point number into integral and fractional parts.
	powf	Computes a power of floating-point number.
	sqrtf	Computes the positive square root of a floating-point number.
	ceilf	Returns the smallest integral value not less than the given floating-point number.

Type	Definition Name	Description
Function	<code>fabsf</code>	Computes the absolute value of a floating-point number.
	<code>floorf</code>	Returns the largest integral value not greater than the given floating-point number.
	<code>fmodf</code>	Computes the remainder of 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 depends on the implementation.

#### 2. Range error

A range error occurs if the result of a function cannot be represented as a **float** value. In this case, the value of **ERANGE** is set in **errno**. If the result overflows, the function returns the value of **HUGE\_VAL**, with the same sign as the correct value of the function. If the result underflows, 0 is returned as the return value.

### Notes

- (1) If there is a possibility of a domain error resulting from a `<mathf.h>` function call, it is dangerous to use the resultant value directly. The value of **errno** should always be checked before using the result in such cases.

#### Example:

```

.
.
.
1  x=asinf(a);
2  if (errno==EDOM)
3      printf ("error\n");
4  else
5      printf ("result is : %f\n", x);
.
.
.

```

In line 1, the arc sine value is computed using the **asinf** function. If the value of parameter **a** is outside the domain of the **asinf** function  $[-1.0, 1.0]$ , the **EDOM** value is set in **errno**. Line 2 determines whether a domain error has occurred. If a domain error has occurred, error is output in line 3. If there is no domain error, the arc sine value is output in line 5.

- (2) Whether or not a range error occurs depends on the internal representation format of floating-point number determined by the compiler. For example, if an internal representation format that allows infinity to be represented as a value is used, `<mathf.h>` library functions can be implemented without causing range errors.

## Implementation Define

Item	Compiler Specifications
Value returned by a mathematical function if an input parameter is out of the range	A not-a-number is returned. For details on the format of not-a-number, refer to section 10.1.3, Floating-Point Number Specifications.
Is <b>errno</b> set to the value of macro <b>ERANGE</b> if an underflow error occurs in a mathematical function?	Not specified
Does a range error occur if the second argument in the <b>fmod</b> function is 0?	An range error occurs.

## float acosf(float f)

Description: Computes the arc cosine of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Arc cosine of **f**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: **f** Floating-point number for which arc cosine is to be computed

Example: 

```
#include <mathf.h>
float f, ret;
ret=acosf(f);
```

Error conditions: A domain error occurs for a value of **f** not in the range  $[-1.0, 1.0]$ .

Remarks: The **acosf** function returns the arc cosine in the range  $[0, \pi]$  by the radian.

## float asinf(float f)

Description: Computes the arc sine of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Arc sine of **f**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: **f** Floating-point number for which arc sine is to be computed

Example: 

```
#include <mathf.h>
float f, ret;
ret=asinf(f);
```

Error conditions: A domain error occurs for a value of **f** not in the range  $[-1.0, 1.0]$ .

Remarks: The **asinf** function returns the arc sine in the range  $[-\pi/2, \pi/2]$  by the radian.

## **float atanf(float f)**

Description: Computes the arc tangent of a floating-point number.

Header file: <mathf.h>

Return values: Arc tangent of **f**

Parameters: **f** Floating-point number for which arc tangent is to be computed

Example:

```
#include <mathf.h>
float f, ret;
ret=atanf(f);
```

Remarks: The **atanf** function returns the arc tangent in the range  $(-\pi/2, \pi/2)$  by the radian.

## float atan2f(float y, float x)

Description: Computes the arc tangent of the division of two floating-point numbers.

Header file: <mathf.h>

Return values: Normal: Arc tangent value when **y** is divided by **x**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: **x** Divisor  
**y** Dividend

Example: 

```
#include <mathf.h>
float x, y, ret;
ret=atan2f(y, x);
```

Error conditions: A domain error occurs if the values of both **x** and **y** are 0.0.

Remarks: The **atan2f** function returns the arc tangent in the range  $(-\pi, \pi]$  by the radian. The meaning of the **atan2f** function is illustrated in figure 10.6. 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  $\pi$ . If **x** = 0.0, the result is  $\pm\pi/2$ , depending on whether **y** is positive or negative.

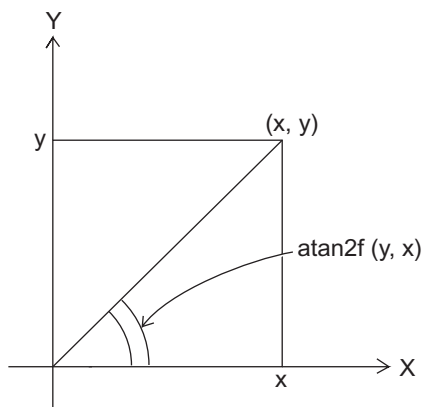


Figure 10.6 Meaning of atan2f Function

## **float cosf(float f)**

Description: Computes the cosine of a floating-point radian value.

Header file: <mathf.h>

Return values: Cosine of **f**

Parameters: **f** Radian value for which cosine is to be computed

Example:

```
#include <mathf.h>
float f, ret;
ret=cosf(f);
```

## **float sinf(float f)**

Description: Computes the sine of a floating-point radian value.

Header file: <mathf.h>

Return values: Sine of **f**

Parameters: **f** Radian value for which sine is to be computed

Example:

```
#include <mathf.h>
float f, ret;
ret=sinf(f);
```

### **float tanf(float f)**

Description: Computes the tangent of a floating-point radian value.

Header file: <mathf.h>

Return values: Tangent of **f**

Parameters: **f** Radian value for which tangent is to be computed

Example:

```
#include <mathf.h>
float f, ret;
ret=tanf(f);
```

### **float coshf(float f)**

Description: Computes the hyperbolic cosine of a floating-point number.

Header file: <mathf.h>

Return values: Hyperbolic cosine of **f**

Parameters: **f** Floating-point number for which hyperbolic cosine is to be computed

Example:

```
#include <mathf.h>
float f, ret;
ret=coshf(f);
```



## **float sinh(float f)**

Description: Computes the hyperbolic sine of a floating-point number.

Header file: <mathf.h>

Return values: Hyperbolic sine of **f**

Parameters: **f** Floating-point number for which hyperbolic sine is to be computed

Example: 

```
#include <mathf.h>
float f, ret;
ret=sinh(f);
```

## **float tanh(float f)**

Description: Computes the hyperbolic tangent of a floating-point number.

Header file: <mathf.h>

Return values: Hyperbolic tangent of **f**

Parameters: **f** Floating-point number for which hyperbolic tangent is to be computed

Example: 

```
#include <mathf.h>
float f, ret;
ret=tanh(f);
```

## **float expf(float f)**

Description: Computes the exponential function of a floating-point number.

Header file: <mathf.h>

Return values: Exponential value of **f**

Parameters: **f** Floating-point number for which exponential function is to be computed

Example: 

```
#include <mathf.h>
float f, ret;
ret=expf(f);
```

## **float frexpf(float value, float ret)**

Description: Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.

Header file: <mathf.h>

Return values: If value is 0.0: 0.0  
If value is not 0.0: Value of ret defined by **ret \* 2<sup>(*e*)</sup> = value**

Parameters: **value** Floating-point number to be broken into a [0.5, 1.0) value and a power of 2  
**e** Pointer to storage area that holds power-of-2 value

Example: 

```
#include <mathf.h>
float ret, value;
int *e
ret=frexpf(value, e);
```

Remarks: The **frexpf** function breaks a 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 **e**.

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 **e** and the value of **ret** are both 0.0.

### **float ldexpf (float ret, int f)**

Description: Multiplies a floating-point number by a power of 2.

Header file: <mathf.h>

Return values: Result of  $e * 2^f$  operation

Parameters: e Floating-point number to be multiplied by a power of 2  
f Power-of-2 value

Example: 

```
#include <mathf.h>
float ret, e;
int f;
ret=ldexpf(e, f);
```

### **float logf(float f)**

Description: Computes the natural logarithm of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Natural logarithm of **f**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: f Floating-point number for which natural logarithm is to be computed

Example: 

```
#include <mathf.h>
float f, ret;
ret=logf(f);
```

Error conditions:

A domain error occurs if **f** is negative.

A range error occurs if **f** is 0.0.

## **float log10f(float f)**

Description: Computes the base-ten logarithm of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Base-ten logarithm of **f**  
Abnormal: In case of domain error: Returns not-a-number.

Parameters: **f** Floating-point number for which base-ten logarithm is to be computed

Example: 

```
#include <mathf.h>
float f, ret;
ret=log10f(f);
```

Error conditions:

A domain error occurs if **f** is negative.

A range error occurs if **f** is 0.0.

## **float modff(float a, float \*b)**

Description: Breaks a floating-point number into integral and fractional parts.

Header file: <mathf.h>

Return values: Fractional part of **a**

Parameters: **a** Floating-point number to be broken into integral and fractional parts  
**b** Pointer indicating storage area that stores integral part

Example: 

```
#include <mathf.h>
float a, *b, ret;
ret=modff(a, b);
```

## **float powf(float x, float y)**

Description:      Computes a power of a floating-point number.

Header file:      <mathf.h>

Return values:    Normal:      Value of **x** raised to the power **y**  
                  Abnormal:    In case of domain error: Returns not-a-number.

Parameters:      **x**              Value to be raised to a power  
                  **y**              Power value

Example:          

```
#include <mathf.h>
float x, y, ret;
ret=powf(x, y);
```

Error conditions:  
                  A domain error occurs if **x** is 0.0 and **y** is 0.0 or less, or if **x** is negative and **y** is not an integer.

## **float sqrtf(float f)**

Description:      Computes the positive square root of a floating-point number.

Header file:      <mathf.h>

Return values:    Normal:      Positive square root of **f**  
                  Abnormal:    In case of domain error: Returns not-a-number.

Parameters:      **f**              Floating-point number for which positive square root is to be computed

Example:          

```
#include <mathf.h>
float f, ret;
ret=sqrtf(x, y);
```

Error conditions:  
                  A domain error occurs if **f** is negative.

### **float ceilf(float f)**

Description: Returns the smallest integral value not less than the given floating-point number.

Header file: <mathf.h>

Return values: Smallest integral value not less than **f**

Parameters: **f** Floating-point number for which smallest integral value not less than that number is to be computed

Example: 

```
#include <mathf.h>
float f, ret;
ret=ceilf(f);
```

Remarks: The **ceilf** function returns the smallest integral value not less than **f**, expressed as a **float**. Therefore, if **f** is negative, the value after truncation of the fractional part is returned.

### **float fabsf(float f)**

Description: Computes the absolute value of a floating-point number.

Header file: <mathf.h>

Return values: Absolute value of **f**

Parameters: **f** Floating-point number for which absolute value is to be computed

Example: 

```
#include <mathf.h>
float f, ret;
ret=fabsf(f);
```

## float floorf(float f)

Description: Returns the largest integral value not greater than the given floating-point number.

Header file: <mathf.h>

Return values: Largest integral value not greater than **f**

Parameters: **f** Floating-point number for which largest integral value not greater than that number is to be computed

Example:

```
#include <mathf.h>
float f, ret;
ret=floorf(f);
```

Remarks: The **floorf** function returns the largest integral value not greater than **f**, expressed as a **float**. Therefore, if **f** is negative, the value after rounding-up of the fractional part is returned.

## float fmodf(float x, float y)

Description: Computes the remainder of division of two floating-point numbers.

Header file: <mathf.h>

Return values: When **y** is 0.0: **x**  
When **y** is not 0.0: Remainder of division of **x** by **y**

Parameters: **x** Dividend  
**y** Divisor

Example:

```
#include <mathf.h>
float x, y, ret;
ret=fmodf(x, y);
```

Remarks: In the **fmodf** function, the relationship between parameters **x** and **y** and return value **ret** is as follows:

$$x = y * i + \text{ret} \text{ (where } i \text{ is an integer)}$$

The sign of return value **ret** is the same as the sign of **x**.

If the quotient of **x/y** cannot be expressed, the value of the result is undefined.

## <setjmp.h>

Supports transfer of control between functions.

The following macros are implementation-defined.

Type	Definition Name	Description
Type (macro)	jmp_buf	Indicates the type name corresponding to a storage area for storing information that enables transfer of control between functions.
Function	setjmp	Saves the calling environment defined by jmp_buf of the currently executing function in the specified storage area.
	longjmp	Restores the function calling 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 calling environment of the current function. The location in the program that called the **setjmp** function can subsequently be returned to by calling the **longjmp** function. An example of how transfer of control between functions is supported using the **setjmp** and **longjmp** functions is shown below.



### Example:

```
1  #include <stdio.h>
2  #include <setjmp.h>
3  jmp_buf env;
4  void main( )
5  {
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 the value (1) specified by the second parameter of the **longjmp** function. As a result, execution proceeds from line 9.

## **int setjmp(jmp\_buf env)**

**Description:** Saves the calling environment of the currently executing function in the specified storage area.

**Header file:** <setjmp.h>

**Return values:** When **setjmp** function is called : 0  
On return from **longjmp** function: Nonzero

**Parameters:** env            Pointer to storage area in which calling environment is saved

**Example:**

```
#include <setjmp.h>
int ret;
jmp_buf env;
ret=setjmp(env);
```

**Remarks:** The calling 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 calling 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.

**void longjmp(jmp\_buf env, int ret)**

**Description:** Restores the function calling environment saved by the **setjmp** function, and transfers control to the program location at which the **setjmp** function was called.

**Header file:** <setjmp.h>

**Parameters:**

<b>env</b>	Pointer to storage area in which calling environment was saved
<b>ret</b>	Return code to <b>setjmp</b> function

**Example:**

```
#include <setjmp.h>
int ret;
jmp_buf env;
longjmp(env, ret);
```

**Remarks:** The **longjmp** function restores from the storage area specified by **env** the function calling 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 **longjmp** function parameter **ret** 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.

## <stdarg.h>

Enables referencing of variable arguments for functions with such arguments.

The following macros are implementation-defined.

Type	Definition Name	Description
Type (macro)	va_list	Indicates the type of variables used in common by the va_start, va_arg, and va_end macros in order to reference variable arguments.
Function (macro)	va_start	Executes initialization processing for performing variable argument referencing.
	va_arg	Enables referencing of the argument following the argument currently being referenced for a function with variable arguments.
	va_end	Terminates referencing of the arguments of a function with variable arguments.

An example of a program using the macros defined by this standard include file is shown below.

### Example:

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  extern void prlist(int count, ...);
5
6  void main( )
7  {
8      prlist(1, 1);
9      prlist(3, 4, 5, 6);
10     prlist(5, 1, 2, 3, 4, 5);
11 }
12
13 void prlist(int count, ...)
14 {
15     va_list ap;
16     int i;
17
18     va_start(ap, count);
19     for(i=0; i<count; i++)
20         printf("%d", va_arg(ap, int));
21     putchar('\n');
22     va_end(ap);
23 }
```

### **Explanation**

In this example, the number of data items to be output is specified in the first argument, and function **prlist** is implemented, outputting that number of subsequent arguments.

In line 18, the variable argument reference is initialized by **va\_start**. Each time an argument is output, the next argument is referenced by the **va\_arg** macro (line 20). In the **va\_arg** macro, the type name of the argument (in this case, **int** type) is specified in the second argument.

When argument referencing ends, the **va\_end** macro is called (line 22).

## **void va\_start(va\_list ap, parmN)**

Description: Executes initialization processing for referencing variable parameters.

Header file: <stdarg.h>

Parameters: ap Variable for accessing variable parameters

parmN Identifier of rightmost argument

Example:

```
#include <stdarg.h>
void func(int count,...){
    va_list ap;
    va_start(ap, count);
}
```

Remarks: The **va\_start** macro initializes **ap** for subsequent use by the **va\_arg** and **va\_end** macros.

The parameter **parmN** is the identifier of the rightmost parameter in the parameter list in the external function definition (the one just before the , ...).

To reference variable nameless arguments, the **va\_start** macro call must be executed first of all.

## **type va\_arg(va\_list ap, type)**

**Description:** Enables referencing of the argument following the argument currently being referenced for a function with variable arguments.

**Header file:** <stdarg.h>

**Return values:** Parameter value

**Parameters:** ap                Variable for accessing variable parameters

type                Type of parameter to be accessed

**Example:**

```
#include <stdarg.h>
va_list ap;
int ret;
    ret=va_arg(ap, int);
```

**Remarks:** A variable of the **va\_list** type initialized by the **va\_start** macro is specified in the first parameter. The value of **ap** is updated each time **va\_arg** is used, and as a result variable parameters are returned sequentially as return values of this macro.

Specify the type of the argument to be referenced at the **type** location in the calling procedure.

The **ap** parameter must be the same as the **ap** initialized by **va\_start**.

It will not be possible to reference the parameters correctly if a type for which the size is changed by type conversion is specified when **char** type, **unsigned char** type, **short** type, **unsigned short** type, or **float** type in the function argument is specified as the type of **type**. If this kind of **type** is specified, correct operation is not guaranteed.

## **void va\_end(va\_list ap)**

Description: Terminates referencing of the arguments of a function with variable arguments.

Header file: <stdarg.h>

Parameters: ap                    Variable for accessing variable arguments

Example:            #include <stdarg.h>  
                     va\_list ap;  
                     va\_end(ap);

Remarks:           The **ap** parameter 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.



## <stdio.h>

Performs processing relating to input/output of stream input/output file.

The following macros are all implementation-defined.

Type	Definition Name	Description
Constant (macro)	FILE	Indicates a structure type that stores various control information including a pointer to the buffer (required for stream input/output processing), an error indicator, and an end-of-file indicator.
	_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 <sup>*1</sup>	Indicates the size of an array large enough to store a string literal 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 <sup>*1</sup>	Indicates the number of files for which simultaneous opening is guaranteed by the implementation.
	TMP_MAX <sup>*1</sup>	Indicates the minimum number of unique file names that shall be generated by the tmpnam function.
	stderr	Indicates the file pointer for the standard error file.
	stdin	Indicates the file pointer for the standard input file.
	stdout	Indicates the file pointer for 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.

Note: 1. Undefined in this implementation.

Type	Definition Name	Description
Function	setbuf	Defines and sets a stream input/output buffer area on the user program side.
	setvbuf	Defines and sets a stream input/output buffer area on the user program side.
	fprintf	Outputs data to a stream input/output file according to a format.
	fscanf	Inputs data from a stream input/output file and converts it according to a format.
	printf	Converts data according to a format and outputs it to the standard output file (stdout).
	scanf	Inputs data from the standard input file (stdin) and converts it according to a format.
	sprintf	Converts data according to a format and outputs it to the specified area.
	sscanf	Inputs data from the specified storage area and converts it according to a format.
	vfprintf	Outputs a variable parameter list to the specified stream input/output file according to a format.
	vprintf	Outputs a variable parameter list to the standard output file according to a format.
	vsprintf	Outputs a variable parameter list to the specified storage 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.

Type	Definition Name	Description
Function	ftell	Obtains the current read/write position in a stream input/output file.
	rewind	Shifts the current read/write position in a stream input/output file to the beginning of the file.
	clearerr	Clears the error state of a stream input/output file.
	feof	Tests for the end of a stream input/output file.
	ferror	Tests for stream input/output file error state.
	perror	Outputs an error message corresponding to the error number to the standard error file (stderr).

## Implementation Define

Item	Compiler Specifications
Does the last line of the input text require a line feed character indicating end?	Not specified. Depends on the low-level interface routine specifications.
Are the space characters immediately before the carriage return character read?	
Number of null characters added to data written in the binary file	
Initial value of file position specifier in the addition mode	
Is a file data lost following text file input?	
File buffering specifications	
Does a file with file length 0 exist?	
File name configuration rule	
Can the same files be opened simultaneously?	
Output format of the %p format conversion in the fprintf function	Hexadecimal representation.
Input data representation of the %p format conversion in the fscanf function. The meaning of conversion character '–' in the fscanf function	Hexadecimal representation. If '–' is not the first or last character or '–' does not follow '^', the compiler indicates the range from the previous character to the following character.
Value of errno specified by the fgetpos or ftell function	The fgetpos function is not supported. The errno value in the ftell function is not specified here. It depends on the low-level interface routine.
Output format of messages generated by the perror function	See (a) below for the output message format.
calloc, malloc, or realloc function operation when the size is 0.	The 0-byte area is allocated.

- (a) The output format of **perror** function is  
<character string>:<error message for the error number specified in error>
- (b) Table 10.34 shows the format when displaying the floating-point infinity and not-a-number in **printf** and **fprintf** functions.

**Table 10.34 Display Format of Infinity and Not-a-Number**

Value	Display Format
Positive infinity	++++++
Negative infinity	-----
Not-a-number	* * * * *

An example of a program that performs a series of input/output processing operations for a stream input/output file is shown in the following.

## Example

```
1  #include <stdio.h>
2
3  void main( )
4  {
5      int c;
6      FILE *ifp, *ofp;
7
8      if ((ifp=fopen("INPUT.DAT", "r"))==NULL){
9          fprintf(stderr, "cannot open input file\n");
10         exit(1);
11     }
12     if ((ofp=fopen("OUTPUT.DAT", "w"))==NULL){
13         fprintf(stderr, "cannot open output file\n");
14         exit(1);
15     }
16     while ((c=getc(ifp))!=EOF)
17         putc(c, ofp);
18     fclose(ifp);
19     fclose(ofp);
20 }
```

## Explanation

This program copies the contents of file INPUT.DAT to file OUTPUT.DAT.

Input file INPUT.DAT is opened by the **fopen** function in line 8, and output file OUTPUT.DAT is opened by the **fopen** function in line 12. If opening fails, NULL is returned as the return value of the **fopen** function, an error message is output, and the program is terminated.

If the **fopen** function ends normally, pointers to the data (**FILE** type) that stores information on the opened files are returned; these are set in variables **ifp** and **ofp**.

After successful opening, input/output is performed using these **FILE** type data items.

When file processing ends, the files are closed with the **fclose** function.

## **int fclose(FILE \*fp)**

Description: Closes a stream input/output file.

Header file: <stdio.h>

Return values: Normal: 0  
Abnormal: Nonzero

Parameters: fp File pointer

Example:

```
#include <stdio.h>
FILE *fp;
int ret;
ret=fclose(fp);
```

Remarks: The **fclose** function closes the stream input/output file indicated by file pointer **fp**.

If the output file of the stream input/output file is open and data that is not output remains in the buffer, that data is output to the file before it is closed.

If the input/output buffer was automatically allocated by the system, it is cancelled.

## **int fflush(FILE \*fp)**

Description: Outputs stream input/output file buffer contents to the file.

Header file: <stdio.h>

Return values: Normal: 0  
Abnormal: Nonzero

Parameters: fp File pointer

Example:

```
#include <stdio.h>
FILE *fp;
int ret;
ret=fflush(fp);
```

Remarks: When an 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 an input file is open, the **ungetc** function specification is invalid.



## FILE \*fopen(const char \*fname, const char \*mode)

Description: Opens a stream input/output file under the specified file name.

Header file: <stdio.h>

Return values: Normal: File pointer indicating file information on opened file  
Abnormal: NULL

Parameters: fname Pointer to string indicating file name  
mode Pointer to string indicating file access mode

Example:

```
#include <stdio.h>
FILE *ret;
const char *fname, *mode;
ret=fopen(fname, mode);
```

Remarks: The **fopen** function opens the stream input/output file whose file name is the string pointed to by **fname**. If a file that does not exist is opened in write mode or addition 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 addition mode, write processing is performed from the end-of-file position. When a file is opened in update mode, both input and output processing can be performed on the file. However, input cannot directly follow output without intervening execution of the **fflush**, **fseek**, or **rewind** function. Similarly, output cannot directly follow input without intervening execution of the **fflush**, **fseek**, or **rewind** function.

A string indicating the opening method may be added after the string indicating the file access mode.

## **FILE \*freopen(const char \*fname, const char \*mode, FILE \*fp)**

**Description:** Closes a currently open stream input/output file and reopens a new file under the specified file name.

**Header file:** <stdio.h>

**Return values:** Normal: fp  
Abnormal: NULL

**Parameters:** fname Pointer to string indicating new file name  
mode Pointer to string indicating file access mode  
fp File pointer of currently open stream input/output file

**Example:**

```
#include <stdio.h>
const char *fname, *mode;
FILE *ret, *fp;
ret=freopen(fname, mode, fp);
```

**Remarks:** The **freopen** function first closes the stream input/output file indicated by file pointer **fp** (the following processing is carried out even if this close processing is unsuccessful). Next, the **freopen** function opens the file indicated by file name **fname** for stream input/output, reusing the **FILE** structure pointed to by **fp**.

The **freopen** function is useful when there is a limit on the number of files being opened at one time.

The **freopen** function normally returns the same value as **fp**, but returns NULL if an error occurs.

**void setbuf (FILE \*fp, char buf[BUFSIZ])**

Description: Defines and sets a stream input/output buffer area by the user program.

Header file: <stdio.h>

Parameters: fp                File pointer  
              buf             Pointer to buffer area

Example:        

```
#include <stdio.h>
FILE *fp;
char buf[BUFSIZ];
                 setbuf(fp, buf);
```

Remarks:        The **setbuf** function defines the storage area pointed to by **buf** so that it can be used as an input/output buffer area for the stream input/output file indicated by file pointer **fp**. As a result, input/output processing is performed using a buffer area of size **BUFSIZ**.

## int setvbuf(FILE \*fp, char \*buf, int type, size\_t size)

Description: Defines and sets a stream input/output buffer area by the user program.

Header file: <stdio.h>

Return values: Normal: 0  
Abnormal: Nonzero

Parameters: fp File pointer  
buf Pointer to buffer area  
type Buffer management method  
size Size of buffer area

Example:

```
#include <stdio.h>
FILE *fp;
char *buf;
int type, ret;
size_t size;
ret=setvbuf(fp, buf, type, size);
```

Remarks: The **setvbuf** function defines the storage area pointed to by **buf** so that it can be used as an input/output buffer area for the stream input/output file indicated by file pointer **fp**.

There are three ways of using this buffer area, as follows:

- (1) When **\_IOFBF** is specified for **type**  
Input/output is fully buffered.
- (2) When **\_IOLBF** is specified for **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.
- (3) When **\_IONBF** is specified for **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 opened stream input/output file is closed. Also, the **setvbuf** function must be used between opening of the stream input/output file and execution of input/output processing,

**int fprintf(FILE \*fp, const char \*control[, arg...])**

Description: Outputs data to a stream input/output file according to the format.

Header file: <stdio.h>

Return values: Normal: Number of characters converted and output  
Abnormal: Negative value

Parameters: fp File pointer  
control Pointer to string indicating format  
arg,... List of data to be output according to format

Example: 

```
#include <stdio.h>
FILE *fp;
const char *control="%s";
int ret;
char buffer[]="Hello World\n";
ret=fprintf(fp, control, buffer);
```

Remarks: The **fprintf** function converts and edits argument **arg** according to the string that indicates 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 data items converted and output, or a negative value if an error occurs.

The format specifications are shown below.

#### (1) Overview of formats

The character string that represents the format is made up of two kinds of string.

##### (a) Ordinary characters

A character other than a conversion specification shown in (b) is output unchanged.

##### (b) Conversion specifications

A conversion specification is a string beginning with % that specifies the conversion method for the following argument. The conversion specifications format conforms to the following rules:

% [Flag ...]  $\left\{ \begin{array}{l} [*] \\ \text{[Field width]} \end{array} \right\} \left[ \cdot \begin{array}{l} [*] \\ \text{[Precision]} \end{array} \right] \text{[Parameter size specifications]} \text{Conversion string}$

When there is no parameter to be actually output for this conversion specifications, the behavior is not guaranteed. Also, when the number of parameters to be actually output is greater than the conversion specifications, the excess parameters are ignored.

(2) Description of conversion specifications

(a) Flags

Flags specify modifications to the data to be output, such as addition of a sign. The types of flags that can be specified, and their meanings, are shown in table 10.35.

**Table 10.35 Flag Types and Their Meanings**

Type	Meaning
–	If the number of converted data characters is less than the field width, the data will be output left-justified within the field.
+	A plus or minus sign will be prefixed to the result of a signed conversion.
space	If the first character of a signed conversion result is not a sign, a space will be prefixed to the result. If the space and + flags are both specified, the space flag will be ignored.
#	The converted data is to be modified according to the conversion types described in table 10.37.
	(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 cannot be 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 appended 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 "0" characters, not spaces, are prefixed to the output data.

(c) Precision

The precision of the converted data is specified according to the type of conversion, as described in table 10.37.

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.

(i) For d, i, o, u, x, and X conversions

The minimum number of digits in the converted data is specified.

(ii) For e, E, and f conversions

The number of digits after the decimal point in the converted data is specified.

(iii) For g and G conversions

The maximum number of significant digits in the converted data is specified.

(iv) 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 10.37), specifies the size (**short** type, **long** type, or **long double** type) of the data to be converted. In other conversions, this specification is ignored. Table 10.36 shows the types of size specification and their meanings.

**Table 10.36 Parameter Size Specification Types and Meanings**

Type	Meaning
h	In d, i, o, u, x, and X conversions, specifies that the data to be converted is of short type or unsigned short type.
l	In 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	In e, E, f, g, and G conversions, specifies that the data to be converted is of long double type.

(e) Conversion specifier

Specifies the format into which data is to be converted.

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 10.37 shows the conversion specifiers 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 another character is specified, depends on the compiler.



**Table 10.37 Conversion Specifiers and Conversion Methods**

Conversion Specifier	Conversion Type	Conversion Method	Data Type Subject to Conversion	Notes on Precision
d	d conversion	int type data is converted to a signed decimal string. d conversion and i conversion have the same specification.	int type	The precision specification indicates the minimum number of characters output. If the number of converted data characters is less than the field width, the string is prefixed with zeros. If the precision is omitted, 1 is assumed. If conversion and output of data with a value of 0 is attempted with 0 specified as the precision, nothing will be output.
i	i conversion		int type	
o	o conversion	int type data is converted to an unsigned octal string.	int type	
u	u conversion	int type data is converted to an unsigned decimal string.	int type	
x	x conversion	int type data is converted to unsigned hexadecimal. a, b, c, d, e, and f are used as hexadecimal characters.	int type	
X	X conversion	int type data is converted to unsigned hexadecimal. A, B, C, D, E, and F are used as hexadecimal characters.	int type	
f	f conversion	double type data is converted to a decimal string with the format [–] ddd.ddd.	double type	The precision specification indicates the number of digits after the decimal point. When there are characters after the decimal point, at least one digit is output before the decimal point. When the precision is omitted, 6 is assumed. When 0 is specified as the precision, the decimal point and subsequent characters are not output. The output data is rounded.
e	e conversion	double type data is converted to a decimal string with the format [–] d.ddde±dd. At least two digits are output as the exponent.	double type	The precision specification indicates the number of digits after the decimal point. The format is such that at least one digit is output before the decimal point in the converted characters, and a number of digits equal to the precision are output after the decimal point. When the precision is omitted, 6 is assumed. When 0 is specified as the precision, characters after the decimal point are not output. The output data is rounded.
E	E conversion	double type data is converted to a decimal string with the format [–] d.dddE±dd. At least two digits are output as the exponent.	double type	

**Table 10.37 Conversion Specifiers and Conversion Methods (cont)**

Conversion Specifier	Conversion Type	Conversion Method	Data Type Subject to Conversion	Notes on Precision
g	g conversion	Whether f conversion format output or e conversion (or E conversion) format output is performed is determined by the value to be converted and the precision value that specifies the number of significant digits, and double type data is output. If the exponent of the converted data is less than -4, or larger than the precision that indicates the number of significant digits, conversion to e (or E) format is performed.	double type	The precision specification indicates the maximum number of significant digits in the converted data.
G	(or G conversion)		double type	
c	c conversion	int type data is converted to unsigned char data, with conversion to the character corresponding to that data.	int type	The precision specification is invalid.
s	s conversion	The string pointed to by pointer to char type are output up to the null character or up to the number of characters specified by the precision. (Null characters are not output. Space, horizontal tab, and new line characters are not included in the converted characters.)	Pointer to char type	The precision specification indicates the number of characters to be output. If the precision is omitted, characters are output up to, but not including, the null character in the string pointed to by the data. (Null characters are not output. Space, horizontal tab, and new line characters are not included in the converted characters.)
p	p conversion	Using data as a pointer, conversion is performed to a string of compiler-defined printable characters.	Pointer to void type	The precision specification is invalid.
n	No conversion is performed.	Data is regarded as pointer to int type, and the number of characters output so far is set in the storage area pointed to by that data.	Pointer to int type	
%	No conversion is performed.	% is output.	None	

(f) \* specification for field width or precision

\* can be specified as the field width or precision specification value. In this case, the value of the parameter corresponding to the conversion specification is used as the field width or precision specification value. When this parameter has a negative field width, flag '-' is interpreted as being specified for the positive field width. When the parameter has a negative precision, the precision is interpreted as being omitted.

## **int fscanf(FILE \*fp, const char \*control[, ptr]...)**

**Description:** Inputs data from a stream input/output file and converts it according to a format.

**Header file:** <stdio.h>

**Return values:** Normal: Number of data items successfully input and converted  
Abnormal: If input data ends before input data conversion is performed: EOF

**Parameters:** fp File pointer  
control Pointer to string indicating format  
ptr Pointer to storage area that holds input data

**Example:**

```
#include <stdio.h>
FILE *fp;
const char *control="%d";
int ret,buffer[10];
ret=fscanf(fp, control, buffer);
```

**Remarks:** The **fscanf** function inputs data from the stream input/output file indicated by file pointer **fp**, converts and edits it according to the string indicating 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.

(1) Overview of formats

The string that represents the format is made up of the following three kinds of string.

(a) White-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.

(b) Ordinary characters

If a character that is neither one of the white-space characters listed in (a) nor % is specified, one input data character is input. The input character must match a character specified in the string that represents the format.

- (c) 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 argument. The conversion specification format conforms to the following rules:

% [\*] [Field width] [Converted data size] Conversion string

If there is no pointer to the storage area that holds input data for the conversion specification in the format, the behavior is not guaranteed. Also, if a pointer to a storage area that holds input data remains though the format is exhausted, that pointer is ignored.

- (2) Description of conversion specifications
- (a) \* specification  
Suppresses storage of the input data in the storage area pointed to by the parameter.
- (b) Field width  
The maximum number of characters in the data to be input is specified as a decimal number.
- (c) Converted data size  
For d, i, o, u, x, X, e, E, and f conversions (see table 10.39), specifies the size (**short** type, **long** type, or **long double** type) of the converted data. In other conversions, this specification is ignored. Table 10.38 shows the types of size specification and their meanings.

**Table 10.38 Converted Data Size Specification Types and Meanings**

Type	Meaning
h	For d, i, o, u, x, and X conversions, specifies that the converted data is of short type.
l	For d, i, o, u, x, and X conversions, specifies that the converted data is of long type. For e, E, and f conversions, specifies that the converted data is of double type.
L	For e, E, and f conversions, specifies that the converted data is of long double type.

- (d) Conversion specifier  
The input data is converted according to the type of conversion specified by the conversion specifier. However, processing is terminated if a white-space character is read, if a character for which conversion is not permitted is read, or if the specified field width is exceeded.

**Table 10.39 Conversion Specifiers and Conversion Methods**

Conversion Specifier	Conversion Type	Conversion Method	Data Type Subject to Conversion
d	d conversion	A decimal string is converted to integer type data.	Integer type
i	i conversion	A decimal string with a sign prefixed, or a decimal string with u (U) or l (L) appended is converted to integer type data. A string beginning with 0x (or 0X) is interpreted as hexadecimal, and the string is converted to int type data. A string beginning with 0 is interpreted as octal, and the string is converted to int type data.	Integer type
o	o conversion	An octal string is converted to integer type data.	Integer type
u	u conversion	An unsigned decimal string is converted to integer type data.	Integer type
x	x conversion	A hexadecimal string is converted to integer type data.	Integer type
X	X conversion	There is no difference in meaning between x conversion and X conversion.	
s	s conversion	Characters are converted as a single string until a space, horizontal tab, or new-line character is read. A null character is appended at the end of the string. (The string in which the converted data is set must be large enough to include the null character.)	Character type
c	c conversion	One character is input. The input character is not skipped even if it is a white-space character. To read only non-white-space characters, specify %1S. If the field width is specified, the number of characters equivalent to that specification are read. In this case, therefore, the storage area that holds the converted data must be of the specified size.	char type
e	e conversion	A string indicating a floating-point number is converted to floating-point type data. There is no difference in meaning between the e conversion and E conversion, or between the g conversion and G conversion. The input format is a floating-point number that can be represented by the strtod function.	Floating-point type
E	E conversion		
f	f conversion		
g	g conversion		
G	G conversion		
p	p conversion	A string converted by p conversion in the fprintf function is converted to pointer type data.	Pointer to void type
n	No conversion is performed.	Data input is not performed; the number of data characters input so far is set.	Integer type
[	[ conversion	A sequence of characters is specified after [, followed by ]. This character sequence defines a sequence of characters comprising a string. If the first character of the character sequence is not a circumflex (^), the input data is input as a single string until a character not in this character sequence is first read. If the first character is ^, the input data is input as a single string until a character which is in the character sequence following the ^ is first read. A null character is automatically appended at the end of the input string (so the string in which the converted data is set must be large enough to include the null character).	Character type
%	No conversion is performed.	% is read.	None

If the conversion specifier is a letter not shown in table 10.39, the behavior is not guaranteed. For other characters, the behavior is implementation-defined.

## **int printf(const char \*control[, arg...])**

**Description:** Converts data according to a format and outputs it to the standard output file (stdout).

**Header file:** <stdio.h>

**Return values:** Normal: Number of characters converted and output  
Abnormal: Negative value

**Parameters:** control Pointer to string indicating format  
arg... Data to be output according to format

**Example:**

```
#include <stdio.h>
char *s;
const char *control="%s";
int ret;
char buffer[]="Hellow World\n";
ret=sprintf(fp,control, buffer);
```

**Remarks:** The **printf** function converts and edits parameter **arg** according to the string that indicates 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.

## **int scanf(const char \*control[, ptr...])**

**Description:** Inputs data from the standard input file (stdin) and converts it according to a format.

**Header file:** <stdio.h>

**Return values:** Normal: Number of data items successfully input and converted  
Abnormal: EOF

**Parameters:** control Pointer to string indicating format  
ptr... Pointer to storage area that holds input and converted data

**Example:**

```
#include <stdio.h>
const char *control="%d";
int ret,buffer[10];
ret=scanf(control,buffer);
```

**Remarks:** The **scanf** function inputs data from the standard input file (stdin), converts and edits it according to the string indicating the format pointed to by **control**, and stores the result in the storage area pointed to by **ptr**.

The **scanf** function returns the number of data items successfully input and converted as the return value. EOF is returned if the standard input file ends before the first conversion.

For details of the format specifications, see the description of the **fscanf()** function.

For %e conversion, specify l for **double** type, and specify L for **long double** type. The default type is **float**.

## **int sprintf(char\* s, const char \*control[, arg...])**

Description: Converts data according to a format and outputs it to the specified area.

Header file: <stdio.h>

Return values: Number of characters converted

Parameters:	s	Pointer to storage area to which data is to be output
	control	Pointer to string indicating format
	arg...	Data to be output according to format

Example:

```
#include <stdio.h>
char *s;
const char *control="%s";
int ret;
char buffer[]="Hellow World\n";
ret=sprintf(fp, control, buffer);
```

Remarks: The **sprintf** function converts and edits parameter **arg** according to the string that indicates 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.



## **int sscanf(const char\*s, const char \*control[, ptr...])**

**Description:** Inputs data from the specified storage area and converts it according to a format.

**Header file:** <stdio.h>

**Return values:** Normal: Number of data items successfully input and converted  
Abnormal: EOF

**Parameters:** s Storage area containing data to be input  
control Pointer to string indicating format  
ptr... Pointer to storage area that holds input and converted data

**Example:**

```
#include <stdio.h>
const char *s, *control="%d";
int ret,buffer[10];
    ret=sscanf(s, control, buffer);
```

**Remarks:** The **sscanf** function inputs data from the storage area pointed to by **s**, converts and edits it according to the string indicating 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 if the input data ends before the first conversion.

For details of the format specifications, see the description of the **fscanf( )** function.

## **int vfprintf(FILE \*fp, const char \*control, va\_list arg)**

**Description:** Outputs a variable parameter list to the specified stream input/output file according to a format.

**Header file:** <stdio.h>

**Return values:** Normal: Number of characters converted and output  
Abnormal: Negative value

**Parameters:** fp File pointer  
control Pointer to string indicating format  
arg Argument list

**Example:**

```
#include <stdarg.h>
#include <stdio.h>
FILE *fp;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
    va_list ap;
    int i;
    va_start(ap, count);
    for(i=0;i<count;i++)
        ret=vfprintf(fp, control, ap);
    va_end(ap)
}
```

**Remarks:** The **vfprintf** function sequentially converts and edits a variable parameter list according to the string that indicates 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 if an error occurs.

With 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 argument list, must be initialized beforehand by the **va\_start** and **va\_arg** macros.

## **int vprintf(const char \*control, va\_list arg)**

**Description:** Outputs a variable parameter list to the standard output file (stdout) according to a format.

**Header file:** <stdio.h>

**Return values:** Normal: Number of characters converted and output  
Abnormal: Negative value

**Parameters:** control Pointer to string indicating format  
arg Argument list

**Example:**

```
#include <stdarg.h>
#include <stdio.h>
FILE *fp;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
    va_list ap;
    int i;
    va_start(ap, count);
    for(i=0;i<count;i++)
        ret=vprintf(control, ap);
    va_end(ap);
}
```

**Remarks:** The **vprintf** function sequentially converts and edits a variable parameter list according to the string that indicates 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 if an error occurs.

With 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 argument list, must be initialized beforehand by the **va\_start** and **va\_arg** macros.

## **int vsprintf(char \*s, const char \*control, va\_list arg)**

Description:	Outputs a variable parameter list to the specified storage area according to a format.	
Header file:	<stdio.h>	
Return values:	Normal:	Number of characters converted
	Abnormal:	Negative value
Parameters:	s	Pointer to storage area to which data is to be output
	control	Pointer to string indicating format
	arg	Argument list
Example:	<pre>#include &lt;stdarg.h&gt; #include &lt;stdio.h&gt; #define NUM 128 char str[NUM]; int ret;  void prlist(int count,...){     va_list ap;     int i;     char *s=str;     va_start(ap, count);     for (i=0;i&lt;count;i++){         ret=vsprintf(s,"%d",ap);         va_arg(ap,int);         s+=ret;     } }</pre>	
Remarks:	<p>The <b>vsprintf</b> function sequentially converts and edits a variable parameter list according to the string that indicates the format pointed to by <b>control</b>, and outputs the result to the storage area pointed to by <b>s</b>.</p> <p>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).</p> <p>For details of the format specifications, see the description of the <b>fprintf( )</b> function.</p> <p>Parameter <b>arg</b>, indicating the argument list, must be initialized beforehand by the <b>va_start</b> and <b>va_arg</b> macros.</p>	

## **int fgetc(FILE \*fp)**

Description: Inputs one character from a stream input/output file.

Header file: <stdio.h>

Return values: Normal: End-of-file: EOF  
Otherwise: Input character  
Abnormal: EOF

Parameters: fp File pointer

Example:

```
#include <stdio.h>
FILE *fp;
int ret;
ret=fgetc(fp);
```

Error conditions:

If a read error occurs, the error indicator for that file is set.

Remarks: The **fgetc** function inputs one character from the stream input/output file indicated by file pointer **fp**.

The **fgetc** function normally returns the input character, but returns EOF at end-of-file or if an error occurs. At end-of-file, the end-of-file indicator for that file is set.

## **char \*fgets(char \*s, int n, FILE \*fp)**

Description: Inputs a string from a stream input/output file.

Header file: <stdio.h>

Return values: Normal: End-of-file: NULL  
Otherwise: s  
Abnormal: NULL

Parameters: s Pointer to storage area to which string is input  
n Number of bytes of storage area to which string is input  
fp File pointer

Example: 

```
#include <stdio.h>
char *s, *ret;
int n;
FILE *fp;
    ret=fgets(s, n, fp);
```

Remarks: The **fgets** function inputs a string from the stream input/output file indicated by file pointer **fp** to the storage area pointed to by **s**.

The **fgets** function performs input up to the (n-1)th character or a new-line character, or until end-of-file, and appends a null character at the end of the input string.

The **fgets** function normally returns **s**, the pointer to the storage area to which the string is input, but returns NULL at end-of-file or if an error occurs.

The contents of the storage area pointed to by **s** do not change at end-of-file, but are undefined if an error occurs.

## **int fputc (int c, FILE \*fp)**

Description:      Outputs one character to a stream input/output file.

Header file:      <stdio.h>

Return values:    Normal:      Output character  
                  Abnormal:    EOF

Parameters:      c              Character to be output  
                  fp             File pointer

Example:          

```
#include <stdio.h>
FILE *fp;
int c, ret;
ret=fputc(c, fp);
```

Error conditions:  
                  If a write error occurs, the error indicator for that file is set.

Remarks:        The **fputc** function outputs character **c** to the stream input/output file indicated by file pointer **fp**.

The **fputc** function normally returns **c**, the output character, but returns EOF if an error occurs.

## int fputs (const char \*s, FILE \*fp)

Description: Outputs a string to a stream input/output file.

Header file: <stdio.h>

Return values: Normal: 0  
Abnormal: Nonzero

Parameters: s Pointer to string to be output  
fp File pointer

Example:

```
#include <stdio.h>
const char *s;
int ret;
FILE *fp;
ret=fputs(s, fp);
```

Remarks: The **fputs** function outputs the string up to the character preceding the null character pointed to by **s** 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 if an error occurs.



## **int getc (FILE \*fp)**

Description: Inputs one character from a stream input/output file.

Header file: <stdio.h>

Return values: Normal: End-of-file: EOF  
Otherwise: Input character  
Abnormal: EOF

Parameters: fp File pointer

Example: 

```
#include <stdio.h>
FILE *fp;
int ret;
ret=getc(fp);
```

Error conditions:

If a read error occurs, the error indicator for that file is set.

Remarks: The **getc** function inputs one character from the stream input/output file indicated by file pointer **fp**.

The **getc** function normally returns the input character, but returns EOF at end-of-file or if an error occurs. At end-of-file, the end-of-file indicator for that file is set.

## **int getchar (void)**

Description: Inputs one character from the standard input file (stdin).

Header file: <stdio.h>

Return values: Normal: End-of-file: EOF  
Otherwise: Input character  
Abnormal: EOF

Example: 

```
#include <stdio.h>
int ret;
ret=getchar();
```

Error conditions: If a read error occurs, the error indicator for that file is set.

Remarks: The **getchar** function inputs one character from the standard input file (stdin).  
  
The **getchar** function normally returns the input character, but returns EOF at end-of-file or if an error occurs. At end-of-file, the end-of-file indicator for that file is set.

## **char \*gets (char \*s)**

Description: Inputs a string from the standard input file (stdin).

Header file: <stdio.h>

Return values: Normal: End-of-file: NULL  
Otherwise: s  
Abnormal: NULL

Parameters: s Pointer to storage area to which string is input

Example: 

```
#include <stdio.h>
char *ret, *s;
ret=gets(s);
```

Remarks: The **gets** function inputs a string from the standard input file (stdin) to the storage area starting at **s**.

The **gets** function inputs characters up to end-of-file or until a new-line character is input, and appends a null character instead of a new-line character.

The **gets** function normally returns **s**, the pointer to the storage area to which the string is input, but returns NULL at the end of the standard input file or if 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 undefined if an error occurs.

## **int putc (int c, FILE \*fp)**

Description: Outputs one character to a stream input/output file.

Header file: <stdio.h>

Return values: Normal: Output character  
Abnormal: EOF

Parameters: c Character to be output  
fp File pointer

Example: 

```
#include <stdio.h>
FILE *fp;
int c, ret;
ret=putc(c, fp);
```

Error conditions: If a write error occurs, the error indicator for that file is set.

Remarks: The **putc** function outputs character **c** to the stream input/output file indicated by file pointer **fp**.

The **putc** function normally returns **c**, the output character, but returns EOF if an error occurs.

## **int putchar(int c)**

Description: Outputs one character to the standard output file (stdout).

Header file: <stdio.h>

Return values: Normal: Output character  
Abnormal: EOF

Parameters: c Character to be output

Example: 

```
#include <stdio.h>
int c, ret;
ret=putchar(c);
```

Error conditions:

If a write error occurs, the error indicator for that file is set.

Remarks: The **putchar** function outputs character **c** to the standard output file (stdout).

The **putchar** function normally returns **c**, the output character, but returns EOF if an error occurs.

## **int puts(const char \*s)**

Description: Outputs a string to the standard output file (stdout).

Header file: <stdio.h>

Return values: Normal: 0  
Abnormal: Nonzero

Parameters: s Pointer to string to be output

Example: 

```
#include <stdio.h>
const char *s;
int ret;
ret=puts(s);
```

Remarks: The **puts** function outputs the string pointed to by **s** to the standard output file (stdout). The null character indicating the end of the string is not output, but a new-line character is output instead.

The **puts** function normally returns zero, but returns nonzero if an error occurs.

## **int ungetc (int c, FILE \*fp)**

Description: Returns one character to a stream input/output file.

Header file: <stdio.h>

Return values: Normal: Returned character  
Abnormal: EOF

Parameters: c Character to be returned  
fp File pointer

Example: 

```
#include <stdio.h>
int c, ret;
FILE *fp;
ret=ungetc(c, fp);
```

Remarks: The **ungetc** function returns character **c** to the stream input/output file indicated by file pointer **fp**. Unless the **fflush**, **fseek**, or **rewind** function is called, this returned character will be the next input data.

The **ungetc** function normally returns character **c**, but returns EOF if an error occurs.

The behavior is not guaranteed if 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, if this file position indicator is already positioned at the beginning of the file, its value will be undefined.

**size\_t fread(void \*ptr, size\_t size, size\_t n, FILE \*fp)**

Description: Inputs data from a stream input/output file to the specified storage area.

Header file: <stdio.h>

Return values: If **size** or **n** is 0: 0  
If **size** and **n** are both nonzero: Number of successfully input members

Parameters:	ptr	Pointer to storage area to which data is input
	size	Number of bytes in one member
	n	Number of members to be input
	fp	File pointer

Example:

```
#include <stdio.h>
void *ptr;
size_t size;
size_t n, ret;
FILE *fp;
    ret=fread(ptr, size, n, fp);
```

Remarks: The **fread** function inputs **n** members whose size is specified by **size**, from the stream input/output file indicated by file pointer **fp**, into the storage area pointed to by **ptr**. The file position indicator for the file is advanced by the number of bytes input.

The **fread** function returns the number of members successfully input, which is normally the same as the value of **n**. However, at end-of-file or if an error occurs, the number of members successfully input so far is returned, and so 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.

If the value of **size** or **n** is zero, zero is returned and the contents of the storage area pointed to by **ptr** are unchanged. If an error occurs, or if only some of the members can be input, the file position indicator will be undefined.

**size\_t fwrite(const void \*ptr, size\_t size, size\_t n, FILE \*fp)**

Description: Outputs data from a memory area to a stream input/output file.

Header file: <stdio.h>

Return values: Number of successfully output members

Parameters:	ptr	Pointer to storage area holding data to be output
	size	Number of bytes in one member
	n	Number of members to be output
	fp	File pointer

Example:

```
#include <stdio.h>
const void *ptr;
size_t size;
size_t n, ret;
FILE *fp;

ret=fwrite(ptr, size, n, fp);
```

Remarks: The **fwrite** function outputs **n** members whose size is specified by **size**, from the storage area pointed to by **ptr**, to the stream input/output file indicated by file pointer **fp**. The file position indicator for the file is advanced by the number of bytes output.

The **fwrite** function returns the number of members successfully output, which is normally the same as the value of **n**. However, if an error occurs, the number of members successfully output so far is returned, and so the return value will be less than **n**.

If an error occurs, the file position indicator will be undefined.



## int fseek(FILE \*fp, long offset, int type)

Description: Shifts the current read/write position in a stream input/output file.

Header file: <stdio.h>

Return values: Normal: 0  
Abnormal: Nonzero

Parameters: fp File pointer  
offset Offset from position specified by type of offset  
type Type of offset

Example: 

```
#include <stdio.h>
FILE *fp;
long offset;
int type, ret;
ret=fseek(fp, offset, type);
```

Remarks: The **fseek** function shifts the current read/write position in the stream input/output file indicated by file pointer **fp**, **offset** bytes from the position specified by the type of **offset** (**type**).  
The types of **offset** are shown in table 10.40.  
The **fseek** function normally returns zero, but returns nonzero in response to an invalid request.

**Table 10.40 Types of Offset**

Offset Type	Meaning
SEEK_SET	Shifts to a position offset bytes from the beginning of the file. The value specified by offset must be zero or positive.
SEEK_CUR	Shifts to a position offset bytes from the current position in the file. The shift is toward the end of the file if the value specified by <b>offset</b> is positive, and toward the beginning of the file if negative.
SEEK_END	Shifts to a position offset bytes forward from end-of-file. The value specified by <b>offset</b> must be zero or negative.

In the case of a text file, the type of **offset** must be **SEEK\_SET** and **offset** must be zero or the value returned by the **ftell** function for that file. Note also that calling the **fseek** function cancels the effect of the **ungetc** function.

## long ftell(FILE \*fp)

Description: Obtains the current read/write position in a stream input/output file.

Header file: <stdio.h>

Return values: Current file position indicator position (text file)  
Number of bytes from beginning of file to current position (binary file)

Parameters: fp File pointer

Example:

```
#include <stdio.h>
FILE *fp;
long ret;
ret=ftell(fp);
```

Remarks: The **ftell** function obtains the current read/write position in the stream input/output file indicated by file pointer **fp**.

For a binary file, the **ftell** function returns the number of bytes from the beginning of the file to the current position. For a text file, it returns, as the position of the file position indicator, an implementation-defined value that can be used by the **fseek** function.

If the **ftell** function is used twice for a text file, the difference in the return values will not necessarily represent the actual distance in the file.

## **void rewind(FILE \*fp)**

Description: Shifts the current read/write position in a stream input/output file to the beginning of the file.

Header file: <stdio.h>

Parameters: fp File pointer

Example:

```
#include <stdio.h>
FILE *fp;
rewind(fp);
```

Remarks: The **rewind** function shifts the current read/write position in the stream input/output file indicated by file pointer **fp**, to the beginning of the file.

The **rewind** function clears the end-of-file indicator and error indicator for the file.

Note that calling the **rewind** function cancels the effect of the **ungetc** function.

## **void clearerr(FILE \*fp)**

Description: Clears the error state of a stream input/output file.

Header file: <stdio.h>

Parameters: fp File pointer

Example:

```
#include <stdio.h>
FILE *fp;
clearerr(fp);
```

Remarks: The **clearerr** function clears the error indicator and end-of-file indicator for the stream input/output file indicated by file pointer **fp**.

## int feof(FILE \*fp)

Description: Tests for the end of a stream input/output file.

Header file: <stdio.h>

Return values: End-of-file: Nonzero  
Otherwise: 0

Parameters: fp File pointer

Example:

```
#include <stdio.h>
FILE *fp;
int ret;
ret=feof(fp);
```

Remarks: The **feof** function tests for the end of the stream input/output file indicated by file pointer **fp**.

The **feof** function tests the end-of-file indicator for the specified stream input/output file, and if the indicator is set, returns nonzero to indicate that the file is at its end. If the end-of-file indicator is not set, the **feof** function returns zero to indicate that the file is not yet at its end.

## **int ferror(FILE \*fp)**

Description: Tests for stream input/output file error state.

Header file: <stdio.h>

Return values: If file is in error state: Nonzero  
Otherwise: 0

Parameters: fp File pointer

Example:

```
#include <stdio.h>
FILE *fp;
int ret;
ret=ferror(fp);
```

Remarks: The **ferror** function tests whether the stream input/output file indicated by file pointer **fp** is in the error state.

The **ferror** function tests the error indicator for the specified stream input/output file, and if the indicator is set, returns nonzero to indicate that the file is in the error state. If the error indicator is not set, the **ferror** function returns zero to indicate that the file is not in the error state.

## **void perror(const char \*s)**

Description: Outputs an error message corresponding to the error number to the standard error file (stderr).

Header file: <stdio.h>

Parameters: s Pointer to error message

Example:

```
#include <stdio.h>
const char *s;
perror(s);
```

Remarks: The **perror** function maps **errno** to the error message indicated by **s**, and outputs the message to the standard error file (stderr).

If **s** is not NULL and the string pointed to by **s** is not the 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.

## <no\_float.h>

Provides simplified I/O functions that does not support the conversion of floating-point numbers (%f, %e, %E, %g, and %G). The ROM size can be minimized when inputting/outputting files that do not require floating-point number conversion.

Type	Definition Name	Description
Function	fprintf	Outputs data to the stream input/output file in the specified format.
	fscanf	Inputs data from the stream input/output file and converts data according to the specified format.
	printf	Converts data according to the specified format, and outputs converted data to the standard output file (stdout).
	scanf	Inputs data from the standard input file (stdin), and converts the input data according to the specified format.
	sprintf	Converts data according to the specified format, and outputs the converted data to the specified area.
	sscanf	Inputs data from the specified memory area, and converts the input data according to the specified format.
	vfprintf	Outputs variable number of parameter lists to the specified stream input/output file according to the specified format.
	vprintf	Outputs variable number of parameter lists to the specified standard output file according to the specified format.
	vsprintf	Outputs variable number of parameter lists to the specified memory area according to the specified format.

Declare `#include <no_float.h>` before specifying `#include <stdio.h>`.

The following shows an example.

```
#include <no_float.h>
#include <stdio.h>
void main(void)
{
    printf("Hello\n");
}
```

### Note

If a floating-point number is specified for a function when `#include <no_float.h>` is specified, correct operation at function execution is not guaranteed.

## <stdlib.h>

Defines functions for standard processing of C programs.

The following macros are implementation-defined.

Type	Definition Name	Description
Type (macro)	onexit_t	Indicates the type returned by the function registered by the onexit function and the type of the onexit function return value.
	div_t	Indicates the type of structure of the value returned by the div function.
	ldiv_t	Indicates the type of structure of the value returned by the ldiv function.
Constant (macro)	RAND_MAX	Indicates the maximum of pseudo-random integers generated by the rand function.
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.
	strtod	Converts a number-representing string to a double type floating-point number.
	strtol	Converts a number-representing string to a long type integer.
	rand	Generates pseudo-random integers from 0 to RAND_MAX.
	srand	Sets an initial value of the pseudo-random number series generated by the rand function.
	calloc	Allocates storage areas and clears all bits in the allocated storage areas to 0.
	free	Releases specified storage area.
	malloc	Allocates a storage area.
	realloc	Changes the size of storage area to a specified value.
	bsearch	Performs binary search.
	qsort	Performs sorting.
	abs	Calculates the absolute value of an int type integer.
	div	Carries out division of int type integers and obtains the quotient and remainder.
	labs	Calculates the absolute value of a long type integer.
	ldiv	Carries out division of long type integers and obtains the quotient and remainder.

## **double atof(const char \*nptr)**

Description: Converts a number-representing string to a **double** type floating-point number.

Header file: <stdlib.h>

Return values: Converted data as a **double** type floating-point number

Parameters: nptr            Pointer to a number-representing string to be converted

Example:

```
#include <stdlib.h>
const char *nptr;
double ret;
ret=atof(nptr);
```

Remarks: Data is converted up to the first character that does not fit the floating-point data type.

The **atof** function sets no **errno** even if an error such as an overflow occurs. If an error occurs, the result will be undefined. When there are possibilities of a conversion error, use the **strtod** function.



## **int atoi(const char \*nptr)**

Description: Converts a decimal-representing string to an **int** type integer.

Header file: <stdlib.h>

Return values: Converted data as an **int** type integer

Parameters: nptr            Pointer to a number-representing string to be converted

Example: 

```
#include <stdlib.h>
const char *nptr;
int ret;
ret=atoi(nptr);
```

Remarks: Data is converted up to the first character that does not fit the decimal data type.

The **atoi** function sets no **errno** even if an error such as an overflow occurs. If an error occurs, the result will be undefined. When there are possibilities of a conversion error, use the **strtol** function.

## **long atol(const char \*nptr)**

Description: Converts a decimal-representing string to a **long** type integer.

Header file: <stdlib.h>

Return values: Converted data as a **long** type integer

Parameters: nptr            Pointer to a number-representing string to be converted

Example: 

```
#include <stdlib.h>
const char *nptr;
long ret;
ret=atol(nptr);
```

Remarks: Data is converted up to the first character that does not fit the decimal data type.

The **atol** function sets no **errno** even if an error such as an overflow occurs. If an error occurs, the result will be undefined. When there are possibilities of a conversion error, use the **strtol** function.

## **double strtod(const char \*nptr, char \*\*endptr)**

Description: Converts a number-representing string to a **double** type floating-point number.

Header file: <stdlib.h>

Return values: Normal: If the string pointed by **nptr** is beginning with a character that does not represent a floating-point number: 0  
If the string pointed by **nptr** is beginning 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 to be converted  
If the converted data underflows: 0

Parameters: **nptr** Pointer to a number-representing string to be converted  
**endptr** Pointer to the storage area containing a pointer to the first character that does not represent a floating-point number

Example:

```
#include <stdlib.h>
const char *nptr;
char **endptr;
double ret;
ret=strtod(nptr, endptr);
```

Error conditions: If the converted result overflows or underflows, **ERANGE** is set for **errno**.

Remarks: According to section 10.1.3 (4), Floating-Point Specifications, the **strtod** function converts data, from the first numeral 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 the exponent nor decimal point is found in the data to be converted, it is assumed that the decimal point comes next to the last numeral in the string. In the address pointed by **endptr**, this function sets up a pointer to the first character that does not compose a floating-point number. If some characters that do not compose a floating-point number come before the first numeral, the value of **nptr** is set in this address. If **endptr** is NULL, nothing is set in this address.

## **long strtol(const char \*nptr, char \*\*endptr, int base)**

Description: Converts a number-representing string to a **long** type integer.

Header file: <stdlib.h>

Return values: Normal: If the string pointed by **nptr** is beginning with a character that does not represent an integer: 0  
If the string pointed by **nptr** is beginning 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 to be converted

Parameters: **nptr** Pointer to a number-representing string to be converted  
**endptr** Pointer to the storage area containing a pointer to the first character that does not represent an integer  
**base** Radix of conversion (0 or 2 to 36)

Example:

```
#include <stdlib.h>
long ret;
const char *nptr;
char **endptr;
int base;
ret=strtol(nptr, endptr, base);
```

Error conditions:

If the converted result overflows, **ERANGE** is set for **errno**.

Remarks: The **strtol** function converts data, from the first numeral to the character immediately before the first character that does not represent an integer, into a **long** type integer.

In the address pointed by **endptr**, this 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 numeral, the value of **nptr** is set in this address. If **endptr** is NULL, nothing is set in this address.

If the value of **base** is 0, data is converted according to section 10.1.1 (4), Integers. 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 are corresponded 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) at **base 16** is ignored.

## **int rand (void)**

Description: Generates pseudo-random integers from 0 to **RAND\_MAX**.

Header file: <stdlib.h>

Return values: Pseudo-random integers

Example: 

```
#include <stdlib.h>
int ret;
ret=rand();
```

## **void srand(unsigned int seed)**

Description: Sets an initial value of the pseudo-random number series generated by the **rand** function.

Header file: <stdlib.h>

Parameters: seed            Initial value for pseudo-random number series generation

Example: 

```
#include <stdlib.h>
unsigned int seed;
srand(seed);
```

Remarks: The **srand** function sets up an initial value for pseudo-random number series generated by the **rand** function. While pseudo-random number series generation by the **rand** function is ongoing, if the same initial value is set up again by the **srand** function, the same pseudo-random number series is repeated.

If the **rand** function is called before the **srand** function, 1 is set as the initial value for the pseudo-random number generation.

## **void \*calloc(size\_t nelem, size\_t elsize)**

Description:      Allocates storage areas and clears all bits in the allocated storage areas to 0.

Header file:      <stdlib.h>

Return values:    Normal:      Starting address of allocated storage area  
                  Abnormal:    If storage allocation failed, or if either of the parameter is 0:  
                                  NULL

Parameters:      nelem          Number of elements  
                  elsize        Number of bytes occupied by a single element

Example:          

```
#include <stdlib.h>
size_t nelem, elsize;
void *ret;
ret=calloc(nelem, elsize);
```

Remarks:        The **calloc** function allocates as many storage areas as specified by **nelem**, in as many units of bytes as specified by **elsize**. The function also clears all the bits in the allocated storage areas to 0.

## **void free(void \*ptr)**

Description:      Releases specified storage area.

Header file:      <stdlib.h>

Parameters:      ptr            Address of storage area to release

Example:          

```
#include <stdlib.h>
void *ptr;
free(ptr);
```

Remarks:        The **free** function releases the storage area pointed by **ptr**, to enable reallocation for use. If **ptr** is NULL, the function carries out nothing.

If the storage area attempted to release was not allocated by the **calloc**, **malloc**, or **realloc** function, or if the area has already been released by the **free** or **realloc** function, correct operation is not guaranteed. Operation result of referencing an already released storage area is also undefined.

## **void \*malloc(size\_t size)**

Description:      Allocates a storage area.

Header file:      <stdlib.h>

Return values:    Normal:      Starting address of allocated storage area  
                  Abnormal:    If storage allocation failed, or if **size** is 0: NULL

Parameters:      size            Size in number of bytes of storage area to allocate

Example:          

```
#include <stdlib.h>
size_t size;
void *ret;
ret=malloc(size);
```

Remarks:          The **malloc** function allocates a storage area of a specified number of bytes by **size**.

## **void \*realloc(void \*ptr, size\_t size)**

Description:      Changes the size of a storage area to a specified value.

Header file:      <stdlib.h>

Return values:    Normal:      Starting address of storage area whose size has been changed  
                  Abnormal:    If storage area allocation failed, or if **size** is 0: NULL

Parameters:      ptr            Starting address of storage area to be changed  
                  size            Size of storage area in number of bytes after the change

Example:          

```
#include <stdlib.h>
size_t size;
void *ptr, *ret;
ret=realloc(ptr, size);
```

Remarks:          The **realloc** function changes the size of the storage area specified by **ptr** to the number of bytes specified by **size**. If the newly allocated storage area is smaller than the old one, the contents are left unchanged up to the size of the newly allocated area.

If the storage area pointed by **ptr** was not allocated by the **calloc**, **malloc**, or **realloc** function, or if the area has already been released by the **free** or **realloc** function, correct operation is not guaranteed.

**void \*bsearch(const void \*key, const void \*base, size\_t nmemb, size\_t size,  
int (\*compar)(const void \*, const void \*))**

Description: Performs binary search.

Header file: <stdlib.h>

Return values: If a matching member is found: Pointer to the matching member  
If no matching member is found: NULL

Parameters:

key	Pointer to data to find
base	Pointer to a table to be searched
nmemb	Number of members to be searched
size	Number of bytes of a member to be searched
compar	Pointer to a function that performs comparison

Example:

```
#include <stdlib.h>
const void *key, *base;
size_t nmemb, size;
int (*compar)(const void *, const void *);
void *ret;

ret=bsearch(key, base, nmemb, size, compar);
```

Remarks: The **bsearch** function searches the table specified by **base** for a member that matches the data specified by **key**, by binary search method. The function that performs comparison should receive pointers **p1** (first parameter) and **p2** (second parameter) to two data items to compare, and return the result complying with the specification below.

If  $*p1 < *p2$ , return a negative value.

If  $*p1 == *p2$ , return 0.

If  $*p1 > *p2$ , return a positive value.

Members to be searched must be placed in the ascending order.

**void qsort(const void \*base, size\_t nmemb, size\_t size,  
int (\*compar)(const void \*, const void\*))**

Description: Performs sorting.

Header file: <stdlib.h>

Parameters:

base	Pointer to a table to sort
nmemb	Number of members to sort
size	Number of bytes of a member to sort
compar	Pointer to a function to perform comparison

Example:

```
#include <stdlib.h>
const void *base;
size_t nmemb, size;
int (*compar)(const void *, const void *)
    qsort(base, nmemb, size, compar);
```

Remarks: The **qsort** function sorts out data on the table specified 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 compare, and return the result complying with the specification below.

If **\*p1 < \*p2**, return a negative value.

If **\*p1 == \*p2**, return 0.

If **\*p1 > \*p2**, return a positive value.



## **int abs(int i)**

Description: Calculates the absolute value of an **int** type integer.

Header file: <stdlib.h>

Return values: Absolute value of **i**

Parameters: **i** Integer to calculate the absolute value of

Example:

```
#include <stdlib.h>
int i, ret;
ret=abs(i);
```

Remarks: If the result cannot be expressed as an **int** type integer, correct operation is not guaranteed.

## **div\_t div(int numer, int denom)**

Description: Carries out division of **int** type integers and obtains the quotient and remainder.

Header file: <stdlib.h>

Return values: Quotient and remainder of division of **numer** by **denom**

Parameters: **numer** Dividend  
**denom** Divisor

Example:

```
#include <stdlib.h>
int numer, denom;
div_t ret;
ret=div(numer, denom);
```

## **long labs(long j)**

Description: Calculates the absolute value of a **long** type integer.

Header file: <stdlib.h>

Return values: Absolute value of **j**

Parameters: **j** Integer to calculate the absolute value of

Example: 

```
#include <stdlib.h>
long j;
long ret;
ret=labs(j);
```

Remarks: If the result cannot be expressed as a **long** type integer, correct operation is not guaranteed.

## **ldiv\_t ldiv(long numer, long denom)**

Description: Carries out division of **long** type integers and obtains the quotient and remainder.

Header file: <stdlib.h>

Return values: Quotient and remainder of division of **numer** by **denom**

Parameters: **numer** Dividend  
**denom** Divisor

Example: 

```
#include <stdlib.h>
long numer, denom;
ldiv_t ret;
ret=ldiv(numer, denom);
```

## <string.h>

Defines functions for manipulating character arrays.

Type	Definition Name	Description
Function	memcpy	Copies contents of a source storage area of a specified length to a destination storage area.
	strcpy	Copies contents of a source string including the null character to a destination storage area.
	strncpy	Copies a source string of a specified length to a destination storage area.
	strcat	Concatenates a string after another string.
	strncat	Concatenates a string of a specified length after another string.
	memcmp	Compares two storage areas specified.
	strcmp	Compares two strings specified.
	strncmp	Compares two strings specified for a specified length.
	memchr	Searches a specified storage area for the first occurrence of a specified character.
	strchr	Searches a specified string for the first occurrence of a specified character.
	strcspn	Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are not included in another string specified.
	strpbrk	Searches a specified string for the first occurrence of any character that is included in another string specified.
	strrchr	Searches a specified string for the last occurrence of a specified character.
	strspn	Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are included in another string specified.
	strstr	Searches a specified string for the first occurrence of another string specified.
	strtok	Divides a specified string into some tokens.
	memset	Sets a specified character for a specified number of times at the beginning of a specified storage area.
	strerror	Sets error messages.
	strlen	Calculates the length of a string.
	memmove	Copies the specified size of the contents of a source area to the destination storage area. If part of the source storage area and the destination storage area overlaps, correct copy is performed.

When using functions defined in this standard include file, note the following.

- (1) When a string is to be copied, if the destination area is smaller than the source area, correct operation is not guaranteed.

Implementation Define

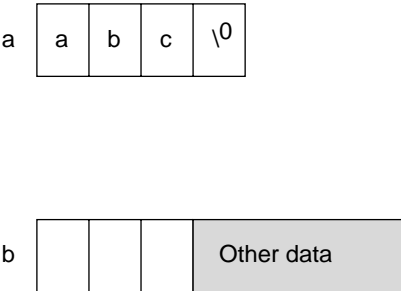
Item	Compiler Specifications
Error message returned by the strerror function	Refer to section 12.3, C Library Error Messages.

Example

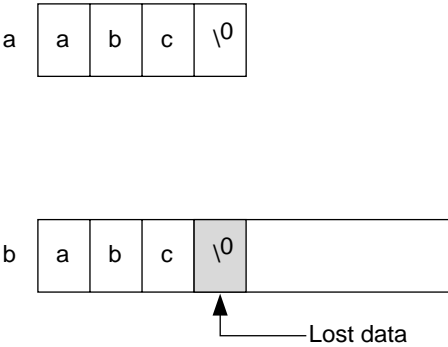
```
char a[]="abc";
char b[3];
.
.
.
strcpy (b, a);
```

In the above example, size of array a (including the null character) is 4 bytes. Copying by **strcpy** overwrites data beyond the boundary of array **b**.

Before copy



After copy

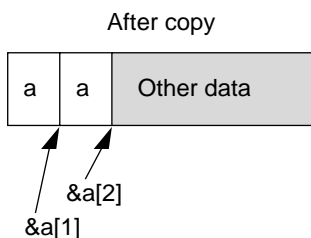
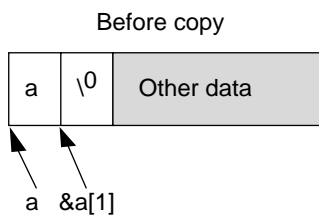


(2) When a string is to be copied, if the source area overlaps the destination area, correct operation is not guaranteed.

#### 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.

## **void \*memcpy(void \*s1, const void \*s2, size\_t n)**

**Description:** Copies contents of a copy source storage area of a specified length to a destination storage area.

**Header file:** <string.h>

**Return values:** **s1** value

**Parameters:**

s1	Pointer to destination storage area
s2	Pointer to source storage area
n	Number of characters to copy

**Example:**

```
#include <string.h>
void *ret, *s1;
const void *s2;
size_t n;
ret=memcpy(s1, s2, n);
```

## **char \*strcpy(char \*s1, const char \*s2)**

**Description:** Copies contents of a source string including the null character to a destination storage area.

**Header file:** <string.h>

**Return values:** **s1** value

**Parameters:**

s1	Pointer to destination storage area
s2	Pointer to source string

**Example:**

```
#include <string.h>
char *s1, *ret;
const char *s2;
ret=strcpy(s1, s2);
```

**char \*strncpy(char \*s1, const char \*s2, size\_t n)**

Description: Copies a source string of a specified length to a destination storage area.

Header file: <string.h>

Return values: **s1** value

Parameters:	s1	Pointer to destination storage area
	s2	Pointer to source string
	n	Number of characters to copy

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
size_t n;
ret=strncpy(s1, s2, n);
```

Remarks: The **strncpy** function copies up to **n** characters in string pointed by **s2** to a storage area pointed by **s1**. If the length of the string specified by **s2** is shorter than **n** characters, the function elongates the string to the length by padding with null characters.

If the length of the string specified by **s2** is longer than **n** characters, the copied string in **s1** storage area ends with a character other than the null character.

**char \*strcat(char \*s1, const char \*s2)**

Description: Concatenates a string after another string.

Header file: <string.h>

Return values: **s1** value

Parameters:	s1	Pointer to the string after which another string is added
	s2	Pointer to the string to add after the other string

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
ret=strcat(s1, s2);
```

Remarks: The **strcat** function concatenates the string specified by **s2** at the end of another string specified by **s1**. The null character indicating the end of the **s2** string is also copied. The null character at the end of the **s1** string is deleted.



**char \*strncat(char \*s1, const char \*s2, size\_t n)**

Description: Concatenates a string of a specified length after another string.

Header file: <string.h>

Return values: **s1** value

Parameters:	s1	Pointer to the string after which another string is added
	s2	Pointer to the string to add after the other string
	n	Number of characters to concatenate

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
size_t n;
ret=strncat(s1, s2, n);
```

Remarks: The **strncat** function concatenates up to **n** characters from the beginning of the string specified by **s2** at the end of another string specified by **s1**. The null character at the end of the **s1** string is replaced by the first character of the **s2** string. A null character is added to the end of the concatenated string.

**int memcmp(const void \*s1, const void \*s2, size\_t n)**

Description: Compares two storage areas specified.

Header file: <string.h>

Return values: If storage area pointed by **s1** > storage area pointed by **s2**: Positive value  
If storage area pointed by **s1** == storage area pointed by **s2**: 0  
If storage area pointed by **s1** < storage area pointed by **s2**: Negative value

Parameters: s1 Pointer to the reference storage area to compare with  
s2 Pointer to the storage area to compare with the reference area  
n Number of characters to compare

Example: 

```
#include <string.h>
const void *s1, *s2;
size_t n;
int ret;
ret=memcmp(s1, s2, n);
```

Remarks: The **memcmp** function compares the contents of the first **n** characters in the storage areas pointed by **s1** and **s2**. The rule of comparison are implementation-defined.

**int strcmp(const char \*s1, const char \*s2)**

Description: Compares two strings specified.

Header file: <string.h>

Return values: If string pointed by **s1** > string pointed by **s2**: Positive value  
If string pointed by **s1** == string pointed by **s2**: 0  
If string pointed by **s1** < string pointed by **s2**: Negative value

Parameters: s1 Pointer to the reference string to compare with  
s2 Pointer to the string to compare with the reference string

Example: 

```
#include <string.h>
const char *s1, *s2;
int ret;
ret=strcmp(s1, s2);
```

Remarks: The **strcmp** function compares the contents of the strings pointed by **s1** and **s2**, and sets up the comparison result as a return value. The rule of comparison are implementation-defined.

**int strncmp(const char \*s1, const char \*s2, size\_t n)**

Description: Compares two strings specified for a specified length.

Header file: <string.h>

Return values: If string pointed by **s1** > string pointed by **s2**: Positive value  
If string pointed by **s1** == string pointed by **s2**: 0  
If string pointed by **s1** < string pointed by **s2**: Negative value

Parameters: s1 Pointer to the reference string to compare with  
s2 Pointer to the string to compare with the reference string  
n Maximum number of characters to compare

Example: 

```
#include <string.h>
const char *s1, *s2;
size_t n;
int ret;
ret=strncmp(s1, s2, n);
```

Remarks: The **strncmp** function compares the contents of the strings pointed by **s1** and **s2**, for up to **n** characters. The rule of comparison are implementation-defined.

**void \*memchr(const void \*s, int c, size\_t n)**

Description: Searches a specified storage area for the first occurrence of a specified character.

Header file: <string.h>

Return values: If the objective character is found: Pointer to the found character  
If the objective character is not found: NULL

Parameters: s Pointer to the storage area to search  
c Character to search for  
n Number of characters to search

Example: 

```
#include <string.h>
const void *s;
int c;
size_t n;
void *ret;
ret=memchr(s, c, n);
```

Remarks: The **memchr** function searches the storage area specified by **s** from the beginning up to **n** characters, looking for the first occurrence of the character specified as **c**. If the **c** character is found, the function returns the pointer to the found character.

## **void \*strchr(const char \*s, int c)**

**Description:** Searches a specified string for the first occurrence of a specified character.

**Header file:** <string.h>

**Return values:** If the objective character is found: Pointer to the found character  
If the objective character is not found: NULL

**Parameters:** s                    Pointer to the string to search  
c                    Character to search for

**Example:**

```
#include <string.h>
const char *s;
int c;
char *ret;
ret=strchr(s, c);
```

**Remarks:** The **strchr** function searches the string specified by **s** looking for the first occurrence of the character specified as **c**. If the **c** character is found, the function returns the found character.

The null character at the end of the **s** string is included in the search objective.

## **size\_t strcspn(const char \*s1, const char \*s2)**

**Description:** Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are not included in another string specified.

**Header file:** <string.h>

**Return values:** Number of consecutive characters at the beginning of the **s1** string that are not included in the **s2** string

**Parameters:** s1                Pointer to the string to check  
s2                Pointer to the string used to check **s1**

**Example:**

```
#include <string.h>
const char *s1, *s2;
size_t ret;
ret=strcspn(s1, s2);
```

**Remarks:** The **strcspn** function checks from the beginning of the string specified by **s1**, and counts the number of consecutive characters that are not included in another string specified by **s2**, and returns that length.

The null character at the end of the **s2** string is not taken as a part of the **s2** string.

## **char \*strpbrk(const char \*s1, const char \*s2)**

**Description:** Searches a specified string for the first occurrence of any character that is included in another string specified.

**Header file:** <string.h>

**Return values:** If the objective character is found: Pointer to the found character  
If the objective character is not found: NULL

**Parameters:** s1                Pointer to the string to search  
s2                Pointer to the string that indicates the characters to search **s1** for

**Example:**

```
#include <string.h>
const char *s1, *s2;
char *ret;
ret=strpbrk(s1, s2);
```

**Remarks:** The **strpbrk** function searches the string specified by **s1** looking for the first occurrence of any character included in the string specified by **s2**. If the searched character is found, the function returns the pointer to the first occurrence.



## **char \*strrchr(const char \*s, int c)**

**Description:** Searches a specified string for the last occurrence of a specified character.

**Header file:** <string.h>

**Return values:** If the objective character is found: Pointer to the found character  
If the objective character is not found: NULL

**Parameters:** s                    Pointer to the string to search  
c                    Character to search for

**Example:**

```
#include <string.h>
const char *s;
int c;
char *ret;
ret=strrchr(s, c);
```

**Remarks:** The **strrchr** function searches the string specified by **s** looking for the last occurrence of the character specified as **c**. If the **c** character is found, the function returns the pointer to the last occurrence of that character.

The null character at the end of the **s** string is included in the search objective.

## **size\_t strstrpn(const char \*s1, const char \*s2)**

**Description:** Checks a specified string from the beginning and counts the number of consecutive characters at the beginning that are included in another string specified.

**Header file:** <string.h>

**Return values:** Number of consecutive characters that are included in the **s2** string at the beginning of the **s1** string

**Parameters:**

s1	Pointer to the string to check
s2	Pointer to the string used to check <b>s1</b>

**Example:**

```
#include <string.h>
const char *s1, *s2;
size_t ret;
ret=strrspn(s1, s2);
```

**Remarks:** The **strstrpn** function checks from the beginning of the string specified by **s1**, and counts the number of consecutive characters that are included in another string specified by **s2**, and returns that length.

**char \*strstr(const char \*s1, const char \*s2)**

Description: Searches a specified string for the first occurrence of another string specified.

Header file: <string.h>

Return values: If the objective string is found: Pointer to the found string  
If the objective string is not found: NULL

Parameters: s1                Pointer to the string to search  
              s2                Pointer to the string to search for

Example: 

```
#include <string.h>
const char *s1, *s2;
char *ret;
ret=strstr(s1, s2);
```

Remarks: The **strstr** function searches the string specified by **s1** looking for the first occurrence of another string specified by **s2**, and returns the pointer to the first occurrence.

## **char \*strtok(char \*s1, const char \*s2)**

Description: Divides a specified string into some tokens.

Header file: <string.h>

Return values: If division into tokens is successful: Pointer to the first token divided  
If division into tokens is unsuccessful: NULL

Parameters: s1                Pointer to the string to divide into some tokens  
s2                Pointer to the string consisting of string dividing characters

Example: 

```
#include <string.h>
char *s1, *ret;
const char *s2;
ret=strtok(s1, s2);
```

Remarks: The **strtok** function should be repeatedly called to divide a string.

### (1) 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 the pointer to the beginning of that token. Otherwise, the function returns NULL.

### (2) Second and subsequent calls

Starting from the next character to the token separated before, the function repeats division at a character included in the string pointed by **s2**. If a token has been separated, the function returns the pointer to the beginning of that token. Otherwise, the function returns NULL.

At the second and subsequent calls, specify NULL for the first parameter.

The string pointed by **s2** can be changed at each call. The null character is added to 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, the 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 argument to consecutively separate tokens from the same string, and repeat calling the **strtok** function.

Consequently, the function separates strings 'b,' 'c,' and 'd.'

**void \*memset(void \*s, int c, size\_t n)**

Description: Sets a specified character for a specified number of times at the beginning of a specified storage area.

Header file: <string.h>

Return values: Value of **s**

Parameters:	<b>s</b>	Pointer to storage area to set characters in
	<b>c</b>	Character to be set
	<b>n</b>	Number of characters to be set

Example:

```
#include <string.h>
void *s, *ret;
int c;
size_t n;
ret=memset(s, c, n);
```

Remarks: The **memset** function sets the character specified by **c** for a number of times specified by **n** to the storage area specified by **s**.

## **char \*strerror(int s)**

Description: Returns an error message corresponding to a specified error number.

Header file: <string.h>

Return values: Pointer to the error message (string) corresponding to the specified error number

Parameters: s                      Error number

Example: 

```
#include <string.h>
char *ret;
int s;
ret=strerror(s);
```

Remarks: The **strerror** function receives an error number specified by **s** and returns an error message corresponding to the number. Contents of error messages are implementation-defined.

If the returned error message is modified, correct operation is not guaranteed.

## **size\_t strlen(const char \*s)**

Description: Calculates the length of a string.

Header file: <string.h>

Return values: Number of characters of the string

Parameters: s                      Pointer to the string to check the length of

Example: 

```
#include <string.h>
const char *s;
size_t ret;
ret=strlen(s);
```

Remarks: The null character at the end of the **s** string is excluded from the string length.

**void \*memmove (void \*s1, const void \*s2, size\_t n)**

**Description:** Copies the specified size of the contents of a source storage area to the destination storage area. If part of the source storage area and the destination storage area overlaps, data is copied to the destination storage area before the overlapped source storage area is overwritten. Therefore, correct copy is enabled.

**Header file:** <string.h>

**Return values:** Value of **s1**

<b>Parameters:</b>	<b>s1</b>	Pointer to the destination storage area
	<b>s2</b>	Pointer to the source storage area
	<b>n</b>	Number of characters to copy

**Example:**

```
#include <string.h>
void *ret, *s1
const void *s2;
size_t n;
ret=memmove(s1, s2, n);
```



## 10.3.2 Embedded C++ Class Libraries

### (1) Overview of Libraries

This section describes the specifications of the embedded C++ class libraries, which can be used as standard libraries in C++ programs. This section gives an overview of the library configuration, and describes the layout and the terms used in this library function description.

#### (a) Library Types

Table 10.41 shows the various library types and the corresponding standard include files.

**Table 10.41 Library Types and Corresponding Standard Include Files**

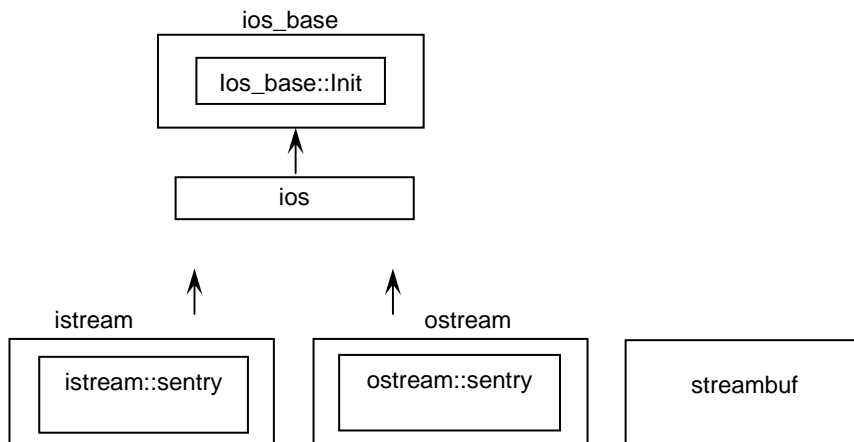
Library Type	Description	Standard Include Files
Stream input/output class	Performs input/output processing.	<ios>, <streambuf>, <istream>,<ostream>, <iostream>,<iomanip>
Memory management	Performs memory allocation and deallocation	<new>
Complex number calculation class	Performs complex number calculation	<complex>
String manipulation	Performs string manipulation	<string>

### (2) Stream Input/Output Class Library

The header files for stream input/output class libraries are as follows.

1. <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.
2. <streambuf>  
Defines functions for the stream buffer.
3. <istream>  
Defines input functions from the input stream.
4. <ostream>  
Defines output functions to the output stream.
5. <iostream>  
Defines input/output functions.
6. <iomanip>  
Defines manipulators with parameters.

The following shows the hierarchy of these classes. Arrows (->) indicate that a derived class refers to a base class. The streambuf class has no hierarchical relation.



The following types are used by stream input/output class libraries.

Type	Definition Name	Description
Type	streamoff	Defined as long type.
	streamsize	Defined as size_t type.
	int_type	Defined as int type.
	pos_type	Defined as long type.
	off_type	Defined as long type.

### (a) ios\_base::Init Class

Type	Definition Name	Description
Variable	init_cnt	Static data member that counts the number of stream input/output objects. The data must be initialized to 0 by a low-level interface.
Function	Init ( )	Constructor
	~ Init ( )	Destructor

1. ios\_base:: Init::Init( )  
Constructor of class Init.  
Increments init\_cnt.
2. ios\_base:: Init::~~ Init ( )  
Destructor of class Init.  
Decrements init\_cnt.

## (b) ios\_base Class

Type	Definition Name	Description
Type	fmtflags	Type that indicates the format control information
	iostate	Type that indicates the stream buffer input/output state
	openmode	Type that indicates the open mode of the file
	seekdir	Type that indicates the seek state of the stream buffer
Variable	fmtfl	Format flag
	wide	Field width
	prec	Precision (number of decimal point digits) at output
	fillch	Fill character
Function	void _ec2p_init_base( )	Initializes the base class
	void _ec2p_copy_base( ios_base& ios_base_dt)	Copies ios_base_dt
	ios_base( )	Constructor
	~ios_base( )	Destructor
	fmtflags flags( ) const	References the format flag (fmtfl)
	fmtflags flags(fmtflags fmtflg)	Sets the result of logical AND of format flag (fmtfl) and fmtflg 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 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 width (wide)
	streamsize width(streamsize wd)	Sets wd as width (wide)

## 1. ios\_base::fmtflags

Defines the format control information relating to input/output processing.

The definition for each bit mask of **fmtflags** is as follows.

```
const ios_base::fmtflags ios_base::boolalpha      = 0x0000;
const ios_base::fmtflags ios_base::skipws         = 0x0001;
const ios_base::fmtflags ios_base::unitbuf        = 0x0002;
const ios_base::fmtflags ios_base::uppercase      = 0x0004;
const ios_base::fmtflags ios_base::showbase       = 0x0008;
const ios_base::fmtflags ios_base::showpoint      = 0x0010;
const ios_base::fmtflags ios_base::showpos        = 0x0020;
const ios_base::fmtflags ios_base::left           = 0x0040;
const ios_base::fmtflags ios_base::right          = 0x0080;
const ios_base::fmtflags ios_base::internal       = 0x0100;
const ios_base::fmtflags ios_base::adjustfield    = 0x01c0;
const ios_base::fmtflags ios_base::dec            = 0x0200;
const ios_base::fmtflags ios_base::oct            = 0x0400;
const ios_base::fmtflags ios_base::hex            = 0x0800;
const ios_base::fmtflags ios_base::basefield      = 0x0e00;
const ios_base::fmtflags ios_base::scientific     = 0x1000;
const ios_base::fmtflags ios_base::fixed          = 0x2000;
const ios_base::fmtflags ios_base::floatfield     = 0x3000;
const ios_base::fmtflags ios_base::_fmtmask       = 0x3fff;
```

## 2. ios\_base::iostate

Defines the input/output state of the stream buffer.

The definition for each bit mask of **iostate** is as follows.

```
const ios_base::iostate ios_base::goodbit      = 0x0;
const ios_base::iostate ios_base::eofbit       = 0x1;
const ios_base::iostate ios_base::failbit      = 0x2;
const ios_base::iostate ios_base::badbit       = 0x4;
const ios_base::iostate ios_base::_statemask    = 0x7;
```

## 3. ios\_base::openmode

Defines open mode of the file.

The definition for each bit mask of **openmode** is as follows.

const ios_base::openmode ios_base::in	= 0x1;	Opens the input file.
const ios_base::openmode ios_base::out	= 0x2;	Opens the output file.
const ios_base::openmode ios_base::ate	= 0x4;	Seeks for eof only once after the file has been opened.
const ios_base::openmode ios_base::app	= 0x8;	Seeks for eof each time the file is written to.
const ios_base::openmode ios_base::trunc	= 0x10;	Opens the file in overwrite mode.
const ios_base::openmode ios_base::binary	= 0x20;	Opens the file in binary mode.

4. `ios_base::seekdir`

Defines the seek state of the stream buffer.

Determines the position to continue the input/output of data in a stream.

The definition for each bit mask of **seekdir** is as follows.

```
const ios_base::seekdir ios_base::beg          = 0x0;
const ios_base::seekdir ios_base::cur          = 0x1;
const ios_base::seekdir ios_base::end          = 0x2;
```

5. `void ios_base::_ec2p_init_base ( )`

The initial settings are as follows.

```
fmtfl = skipws | dec;
wide = 0;
prec = 6;
fillch = ' ';
```

6. `void ios_base::_ec2p_copy_base (ios_base & ios_base_dt)`

Copies **ios\_base\_dt**.

7. `ios_base::ios_base( )`

Constructor of class **ios\_base**.

Calls **Init::Init( )**.

8. `ios_base::~ios_base( )`

Destructor of class **ios\_base**.

9. `ios_base::fmtflags ios_base::flags ( ) const`

References format flag (**fmtfl**).

Return value: Format flag (**fmtfl**)

10. `ios_base::fmtflags ios_base::flags(fmtflags fmtflg)`

Sets **fmtflg**&format flag (**fmtfl**) to format flag (**fmtfl**).

Return value: Format flag (**fmtfl**) before setting

11. `ios_base::fmtflags ios_base::setf (fmtflags fmtflg)`

Sets **fmtflg** to format flag (**fmtfl**).

Return value: Format flag (**fmtfl**) before setting

12. `ios_base::fmtflags ios_base::setf(fmtflags fmtflg, fmtflags mask)`

Sets mask&**fmtflg** to format flag (**fmtfl**).

Return value: Format flag (**fmtfl**) before setting.

13. `void ios_base::unsetf (fmtflags mask)`

Sets ~mask&format flag (**fmtfl**) to format flag (**fmtfl**).

14. `char ios_base::fill ( ) const`  
References fill character (**fillch**).  
Return value: Fill character (**fillch**)
15. `char ios_base::fill(char ch)`  
Sets **ch** as fill character (**fillch**).  
Return value: Fill character (**fillch**) before setting
16. `int ios_base::precision ( ) const`  
References precision (**prec**).  
Return value: Precision (**prec**)
17. `streamsize ios_base::precision(streamsize preci)`  
Sets **preci** as precision (**prec**).  
Return value: Precision (**prec**) before setting
18. `streamsize ios_base::width ( ) const`  
References width (**wide**).  
Return value: Width (**wide**)
19. `streamsize ios_base::width(streamsize wd)`  
Sets **wd** as width (**wide**).  
Return value: Width (**wide**) before setting



### (c) ios Class

Type	Definition Name	Description
Variable	sb	Pointer to streambuf object
	tiestr	Pointer to 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))
	iosstate rdstate( ) const	References the state flag (state)
	void clear(iosstate st=goodbit)	Clears the state flag (state) except for the specified state (st)
	void setstate(iosstate 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 input text matches the requested pattern (state&(badbit failbit))
	ostream* tie( ) const	References the pointer to the ostream object (tiestr)
	ostream* tie(ostream* tstrptr)	Specifies tstrptr as the pointer to the ostream object (tiestr)
	streambuf* rdbuf( ) const	References the pointer (sb) to the streambuf object
	streambuf* rdbuf(streambuf* sbptr)	Specifies sbptr as the pointer (sb) to the streambuf object
	ios & copyfmt(const ios& rhs)	Copies the state flag (state) of rhs

1. `ios::ios ( )`  
 Constructor of class **ios**.  
 Calls **init(0)** and specifies the initial value in the member object.
2. `ios::ios(streambuf *sbptr)`  
 Constructor of class **ios**.  
 Calls **init(sbptr)** and specifies the initial value in the member object.
3. `void ios::init (streambuf *sbptr)`  
 Specifies **sb** in **sbptr**.  
 Specifies **state** and **tiestr** as 0.
4. `virtual ios::~ios( )`  
 Destructor of class **ios**.
5. `ios::operator void* ( ) const`  
 Tests whether an error has been generated (`!state&(badbit|failbit)`).  
 Return value: An error has been generated: false  
                   No error has been generated: true
6. `bool ios::operator! ( ) const`  
 Tests whether an error has been generated (`state&(badbit|failbit)`).  
 Return value: An error has been generated: true  
                   No error has been generated: false
7. `iosstate ios::rdstate ( ) const`  
 References the state flag (**state**).  
 Return value: State flag (**state**)
8. `void ios::clear (iosstate st=goodbit)`  
 Clears the state flag (**state**) except for the specified state (**st**).  
 If the pointer to the **streambuf** object (**sb**) is 0, **badbit** is set to the state flag (**state**).
9. `void ios::setstate (iosstate st)`  
 Specifies the contents of **st** in the state flag (**state**).
10. `bool ios::good ( ) const`  
 Tests whether an error has been generated (`state==goodbit`).  
 Return value: An error has been generated: false  
                   No error has been generated: true

11. `bool ios::eof ( ) const`  
Tests for the end of the input stream (`state&eofbit`).  
Return value: End of the input stream has been reached: `true`  
End of the input stream has not been reached: `false`
12. `bool ios::bad ( ) const`  
Tests whether an error has been generated (`state&badbit`).  
Return value: An error has been generated: `true`  
No error has been generated: `false`
13. `bool ios::fail ( ) const`  
Tests whether the input text matches the requested pattern (`state&(badbit|failbit)`).  
Return value: Does not match the requested pattern: `true`  
Matches the requested pattern: `false`
14. `ostream* ios::tie ( ) const`  
References the pointer to the `ostream` object (**`tiestr`**).  
Return value: Object pointer (**`tiestr`**)
15. `ostream* ios::tie(ostream* tstrptr)`  
Specifies **`tstrptr`** as the pointer to the `ostream` object (**`tiestr`**).  
Return value: **`ostream`** object pointer (**`tiestr`**) before setting
16. `streambuf* ios::rdbuf ( ) const`  
References the pointer to the **`streambuf`** object (**`sb`**).  
Return value: Pointer (**`sb`**) to **`streambuf`** object
17. `streambuf* ios::rdbuf(streambuf* sbptr)`  
Specifies **`sbptr`** as the pointer to the **`streambuf`** object (**`sb`**).  
Return value: Pointer to the **`streambuf`** object (**`sb`**) before setting
18. `ios & ios::copyfmt (const ios & rhs)`  
Copies the state flag (**`state`**) of **`rhs`**.  
Return value: `*this`

#### (d) ios Class Manipulators

Type	Definition Name	Description
Function	ios_base& boolalpha(ios_base& str)	Specifies bool type format
	ios_base& noboolalpha( ios_base& str)	Clears bool type format
	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)	Clears the left side fill mode
	ios_base& right(ios_base& str)	Clears 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 output mode
	ios_base& scientific(ios_base& str)	Specifies the scientific description mode

1. ios\_base& boolalpha(ios\_base& str)  
Specifies bool type format.  
Return value: str
2. ios\_base& noboolalpha(ios\_base& str)  
Clears bool type format.  
Return value: str

3. `ios_base& showbase(ios_base& str)`  
Specifies a mode to prefix 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`
4. `ios_base& noshowbase(ios_base &str)`  
Clears the mode to prefix a radix.  
Return value: `str`
5. `ios_base& showpoint(ios_base & str)`  
Specifies the mode to output decimal point.  
If no precision is specified, six decimal-point (fraction) digits are displayed.  
Return value: `str`
6. `ios_base& noshowpoint(ios_base& str)`  
Clears the mode to output decimal point.  
Return value: `str`
7. `ios_base& showpos(ios_base& str)`  
Specifies the + sign generation mode (adds a + sign to a positive number).  
Return value: `str`
8. `ios_base& noshowpos(ios_base & str)`  
Clears the + sign generation mode.  
Return value: `str`
9. `ios_base& skipws(ios_base& str)`  
Specifies the space skipping mode (skips consecutive spaces).  
Return value: `str`
10. `ios_base& noskipws(ios_base& str)`  
Clears the space skipping mode.  
Return value: `str`
11. `ios_base& uppercase(ios_base& str)`  
Specifies the uppercase letter conversion output mode.  
For a hexadecimal, the radix will be the uppercase letters 0X, and the numeric value letters will be uppercase letters. The exponential representation of a floating-point value will use uppercase letter E.  
Return value: `str`

12. `ios_base::nouppercase(ios_base & str)`  
Clears the uppercase letter conversion output mode.  
Return value: `str`
13. `ios_base::internal(ios_base & str)`  
When data is output in the field width (**wide**) range, it is output in the order of
  1. Sign and radix
  2. Fill character (fill)
  3. Numeric valueReturn value: `str`
14. `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`
15. `ios_base::right(ios_base & str)`  
When data is output in the field width (**wide**) range, it is aligned to the right.  
Return value: `str`
16. `ios_base::dec(ios_base & str)`  
Specifies the conversion radix as the decimal mode.  
Return value: `str`
17. `ios_base::hex(ios_base & str)`  
Specifies the conversion radix as the hexadecimal mode.  
Return value: `str`
18. `ios_base::oct(ios_base & str)`  
Specifies the conversion radix as the octal mode.  
Return value: `str`
19. `ios_base::fixed(ios_base & str)`  
Specifies the fixed-point output mode.  
Return value: `str`
20. `ios_base::scientific(ios_base & str)`  
Specifies the scientific description mode (exponential description).  
Return value: `str`

### (e) streambuf Class

Type	Definition Name	Description
Constant	eof	Indicates the end of 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 to read data.
	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 to read data.
	C_flg_ptr	Pointer to the input/output control flag of the file.
Function	char* _ec2p_getflag( ) const	References a pointer for file input/output control flag.
	char* & _ec2p_gnptr( )	References a pointer to the next position of the buffer from which to read data.
	char* & _ec2p_pnptr( )	References a pointer to the next position of the buffer where data is to be written.
	void _ec2p_bcntplus( )	Increments the length of valid data in the buffer.
	void _ec2p_bcntminus( )	Decrements the length of valid data in 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)	Defines buffer for stream input/output. This function calls setbuf (s,n) <sup>1</sup> .

Type	Definition Name	Description
Function	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 for the input/output stream by using the method specified by way. This function calls seekoff(off,way,which) <sup>1</sup> .
	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) <sup>1</sup> .
	int pubsync( )	Flushes the output stream. This function calls sync( ) <sup>1</sup> .
	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.
	int_type sgetc( )	Reads one character.
	int sgetn(char* s, streamsize n)	Sets n number of characters in the memory area specified by s.
	int_type sputbackc(char c)	Puts back the read position.
	int sungetc( )	Puts back the read position.
	int sputc(char c)	Inserts characters c.
	int_type sputn(const char* s, streamsize n)	Inserts n number of characters specified by s.
	char* eback( ) const	Calculates the start pointer of the input stream.
	char* gptr( ) const	Calculates the next pointer of the input stream.
	char* egptr( ) const	Calculates the end pointer of the input stream.
	void gbump(int n)	Moves the next pointer of the input stream for n.
	void setg( char* gbegin, char* gnext, char* gend)	Assigns each pointer of the input stream.



Type	Definition Name	Description
Function	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 for n.
	void setp(char* pbeg, char* pend)	Specifies each pointer of the output stream.
	virtual streambuf *setbuf(char* s, streamsize n) <sup>1</sup>	For each derived class, a defined operation is executed.
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode=(ios_base::openmode) (ios_base::in   ios_base::out)) <sup>1</sup>	Changes the stream position.
	virtual pos_type seekpos(pos_type sp, ios_base::openmode=(ios_base::openmode) (ios_base::in   ios_base::out)) <sup>1</sup>	Changes the stream position.
	virtual int sync( ) <sup>1</sup>	Flushes the output stream.
	virtual int showmanyc( ) <sup>1</sup>	Calculates the number of valid characters in the input stream.
	virtual streamsize xsgetn(char* s, streamsize n)	Sets n number of characters in the memory area specified by s.
	virtual int_type underflow( ) <sup>1</sup>	Reads one character without moving the stream position.
	virtual int_type uflow( ) <sup>1</sup>	Reads one character of the next pointer.
	virtual int_type pbackfail(int type c = eof) <sup>1</sup>	Puts back the character specified by c.
	virtual streamsize xspn(const char* s, streamsize n)	Inserts n number of characters specified by s.
	virtual int_type overflow(int type c = eof) <sup>1</sup>	Inserts c in the output stream.

Note<sup>1</sup>: This class does not define the processing.

1. `streambuf::streambuf ( )`  
 Constructor.  
 The initial settings are as follows:  
`_B_cnt_ptr = B_beg_ptr = B_next_ptr = B_end_ptr = C_flg_ptr = B_len_ptr = 0`  
`B_beg_pptr = &B_beg_ptr`  
`B_next_pptr = &B_next_ptr`
2. `virtual streambuf::~streambuf ( )`  
 Destructor.
3. `streambuf* streambuf::pubsetbuf (char* s, streamsize n)`  
 Defines the buffer for stream input/output.  
 This function calls **setbuf (s,n)**.  
 Return value: `*this`
4. `pos_type streambuf::pubseekoff (off_type off, ios_base::seekdir way, ios_base::openmode which=(ios_base::openmode)(ios_base::in|ios_base::out))`  
 Moves the position to read or write data for the input/output stream by using the method specified by **way**.  
 This function calls **seekoff(off,way,which)**.  
 Return value: Newly specified stream position
5. `pos_type streambuf::pubseekpos (pos_type sp, ios_base::openmode which=(ios_base::openmode)(ios_base::in | ios_base::out))`  
 Calculates the offset from the beginning of the stream to the current position.  
 Moves the current stream pointer for **sp**.  
 This function calls **seekpos(sp,which)**.  
 Return value: The offset from the beginning of the stream
6. `int streambuf::pubsync ( )`  
 Flushes the output stream.  
 This function calls **sync( )**.  
 Return value: 0
7. `streamsize streambuf::in_avail ( )`  
 Calculates the offset from the end of the input stream to the current position.  
 Return value: If the position where data is read is valid: The offset from the end of the stream to the current position.  
 If the position where data is read is invalid: 0 (**showmanyc( )** is called)

8. `int_type streambuf::snextc ( )`  
Reads one character. If the character read is not eof, the next character is read.  
Return value: If the characters read is not eof: The character read  
If the characters read is eof: eof
9. `int_type streambuf::sbumpc ( )`  
Reads one character and moves the pointer to the next.  
Return value: If the position where data is read is valid: The character read  
If the position where data is read is invalid: eof
10. `int_type streambuf::sgetc ( )`  
Reads one character.  
Return value: If the position where data is read is valid: The character read  
If the position where data is read is invalid: eof
11. `int streambuf::sgetn (char* s, streamsize n)`  
Sets **n** number of characters in the memory area specified by **s**. If an eof is found in the string literal, this setting is terminated.  
Return value: The specified number of characters.
12. `int_type streambuf::sputbackc (char c) ;`  
If the data read position is correct and the put back data of the position is the same as **c**, the read position is put back.  
Return value: If the read position was put back: The value of **c**  
If the read position was not put back: eof
13. `int streambuf::sungetc ( )`  
If the data read position is correct, the read position is put back.  
Return value: If the read position was put back: The value that was put back  
If the read position was not put back: eof
14. `int streambuf::sputc (char c)`  
Inserts characters **c**.  
Return value: If the write position is correct: The value of **c**  
If the write position is incorrect: eof

15. `int_type streambuf::sputn (const char* s, streamsize n)`  
Inserts **n** number of characters specified by **s**.  
If the buffer is smaller than **n**, the number of characters for the buffer size is inserted.  
Return value: The number of characters inserted
16. `char* streambuf::eback ( ) const`  
Calculates the start pointer of the input stream.  
Return value: Start pointer
17. `char* streambuf::gptr ( ) const`  
Calculates the next pointer of the input stream.  
Return value: Next pointer
18. `char* streambuf::egptr ( ) const`  
Calculates the end pointer of the input stream.  
Return value: End pointer
19. `void streambuf::gbump (int n)`  
Moves the next pointer of the input stream for **n**.
20. `void streambuf::setg (char* gbeg, char* gnext, char* gend)`  
The settings for each pointer of the input stream are as follows:  
    `*B_beg_pptr = gbeg;`  
    `*B_next_pptr = gnext;`  
    `B_end_ptr = gend;`  
    `*_B_cnt_ptr = gend-gnext;`  
    `*_B_len_ptr = gend-gbeg;`
21. `char* streambuf::pbase ( ) const`  
Calculates the start pointer of the output stream.  
Return value: Start pointer
22. `char* streambuf::pptr ( ) const`  
Calculates the next pointer of the output stream.  
Return value: Next pointer

23. `char* streambuf::eptr ( ) const`  
Calculates the end pointer of the output stream.  
Return value: End pointer
24. `void streambuf::pbump (int n)`  
Moves the next pointer of the output stream for **n**.
25. `void streambuf::setp (char* pbeg, char* pend)`  
The settings for each pointer of the output stream are as follows:  
`*B_beg_pptr = pbeg;`  
`*B_next_pptr = pbeg;`  
`B_end_ptr = pend;`  
`*_B_cnt_ptr=pend-pbeg;`  
`*_B_len_ptr=pend-pbeg;`
26. `virtual streambuf* streambuf::setbuf (char* s, streamsize n)`  
For each derived class of `streambuf`, a defined operation is executed.  
Return value: `*this` (This class does not define the processing)
27. `virtual pos_type streambuf::seekoff (off_type off, ios_base::seekdir way, ios_base::openmode=(ios_base::openmode)(ios_base::in | ios_base::out))`  
Changes the stream position.  
Return value: (-1) (This class does not define the processing)
28. `virtual pos_type streambuf::seekpos (pos_type off, ios_base::openmode=(ios_base::openmode)(ios_base::in | ios_base::out))`  
Changes the stream position.  
Return value: (-1) (This class does not define the processing)
29. `virtual int streambuf::sync ( )`  
Flushes the output stream.  
Return value: 0 (This class does not define the processing)
30. `virtual int streambuf::showmanyc ( )`  
Calculates the number of valid characters in the input stream.  
Return value: 0 (This class does not define the processing)

31. virtual streamsize streambuf::xsgetn (char\* s, streamsize n)  
Sets **n** number of characters in the memory area specified by **s**.  
If the buffer is smaller than **n**, the numbers of characters for the buffer size is inserted.  
Return value: The number of characters input
32. virtual int\_type streambuf::underflow ( )  
Reads one character without moving the stream position.  
Return value: eof (This class does not define the processing)
33. virtual int\_type streambuf::uflow ( )  
Reads one character of the next pointer.  
Return value: eof (This class does not define the processing)
34. virtual int\_type streambuf::pbackfail (int\_type c=eof)  
Puts back the character specified by **c**.  
Return value: eof (This class does not define the processing)
35. virtual streamsize streambuf::xsputn (const char\* s, streamsize n)  
Inserts **n** number of characters specified by **s**.  
If the buffer is smaller than **n**, the number of characters for the buffer size is inserted.  
Return value: The number of characters inserted
36. virtual int\_type streambuf::overflow (int\_type c=eof)  
Inserts **c** in the output stream.  
Return value: eof (This class does not define the processing)

## (f) istream::sentry Class

Type	Definition Name	Description
Variable	ok_	Whether the current state is input-enabled
Function	sentry (istream& is, bool noskipws= false)	Constructor
	~sentry( )	Destructor
	operator bool( )	References ok_

1. istream::sentry::sentry(istream& is, bool noskipws=\_false)  
Constructor of internal class sentry.  
Return value: If **good()** is non-zero, enables input with or without a format.  
If **tie()** is non-zero, flushes related output stream.
2. istream::sentry::~sentry ( )  
Destructor of internal class sentry
3. istream::sentry::operator bool ( )  
References ok\_.  
Return value: ok\_

## (g) istream Class

Type	Definition Name	Description
Variable	chcount	The number of characters extracted by the input function called last.
Function	int _ec2p_getistr(char* str,unsigned int dig, int mode)	Converts str with the radix specified by dig.
	istream(streambuf* sb)	Constructor.
	virtual ~istream( )	Destructor.
	istream& operator>>(bool& n)	Stores the extracted characters in n.
	istream& operator>>(short& n)	
	istream& operator>>(unsigned short& n)	
	istream& operator>>(int& n)	
	istream& operator>>(unsigned int& n)	
	istream& operator>>(long& n)	
	istream& operator>>(unsigned long& n)	
	istream& operator>>(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 it in p.
	istream& operator >>(streambuf* sb)	Extracts characters and stores them in the memory area specified by sb.
	streamsize gcount( ) const	Calculates chcount (number of characters extracted).
	int_type get( )	Extracts a character.



Type	Definition Name	Description
Function	istream& get(char& c)	Stores the extracted characters in c.
	istream& get(signed char& c)	
	istream& get(unsigned char& c)	
	istream& get(char* s, streamsize n)	Extracts string literals with size n-1 and stores them in the memory area specified by s.
	istream& get(signed char* s, streamsize n)	
	istream& get(unsigned char* s, streamsize n)	
	istream& get(char* s, streamsize n, char delim)	Extracts string literals with size n-1 and stores them in the memory area specified by s. If delim is found in the string literal, input is stopped.
	istream& get(signed char* s, streamsize n, char delim)	
	istream& get(unsigned char* s, streamsize n, char delim)	
	istream& get(streambuf& sb)	Extracts string literals and stores them in the memory area specified by sb.
	istream& get(streambuf& sb, char delim)	Extracts string literals and stores them in the memory area specified by sb. If character delim is found, input is stopped.
	istream& getline(char* s, streamsize n)	Extracts string literals with size n-1 and stores them in the memory area specified by s.
	istream& getline(signed char* s, streamsize n)	
	istream& getline(unsigned char* s, streamsize n)	

Type	Definition Name	Description
Function	istream& getline(char* s, streamsize n, char delim)	Extracts string literals with size n-1 and stores them in the memory area specified by s. If character delim is found, input is stopped.
	istream& getline( signed char* s, streamsize n, char delim)	
	istream& getline( unsigned char* s, streamsize n, char delim)	
	istream& ignore( streamsize n=1, int_type delim=streambuf::eof)	Skips reading the number of characters specified by n. If character delim is found, skipping is stopped.
	int_type peek( )	Seeks for input characters that can be acquired next.
	istream& read(char* s, streamsize n)	Extracts string literals with size n and stores them in the memory area specified by s.
	istream& read(signed char* s, streamsize n)	
	istream& read(unsigned char* s, streamsize n)	
	streamsize readsome(char* s, streamsize n)	Extracts string literals with size n and stores them in the memory area specified by s.
	streamsize readsome(signed char* s, streamsize n)	
	streamsize readsome( unsigned char* s, streamsize n)	
	istream& putback(char c)	Returns a character to the input stream.
	istream& unget( )	Returns the position of the input stream.
	int sync( )	Checks for an input stream. This function calls streambuf::pubsync( ).

Type	Definition Name	Description
Function	pos_type tellg( )	Checks for the input stream position. This function calls streambuf::pubseekoff(0,cur,in).
	istream& seekg(pos_type pos)	Moves the current stream pointer for pos. This function calls streambuf::pubseekpos(pos).
	istream& seekg(off_type off, ios_base::seekdir dir)	Moves the position to read the input stream by using the method specified by dir. This function calls stream::pubseekoff(off,dir).

1. int istream::\_ec2p\_getistr (char\* str, unsigned int dig, int mode)

Converts **str** with the radix specified by **dig**.

Return value: Returns the converted radix.

2. istream::istream (streambuf\* sb)

Constructor of class **istream**.

Calls ios::init(sb).

Specifies chcount=0.

3. virtual istream::~~istream ( )

Destructor of class **istream**.

4. istream& istream::operator>> (bool& n)  
istream& istream::operator>> (short& n)  
istream& istream::operator>> (unsigned short& n)  
istream& istream::operator>> (int& n)  
istream& istream::operator>> (unsigned int& n)  
istream& istream::operator>> (long& n)  
istream& istream::operator>> (unsigned long& n)  
istream& istream::operator>> (float& n)  
istream& istream::operator>> (double& n)  
istream& istream::operator>> (long double& n)

Stores the extracted characters in **n**.

Return value: \*this

5. `istream& istream::operator>> (void*& p)`  
 Converts the extracted characters to a void type and stores them in the memory specified by **p**.  
 Return value: `*this`
  
6. `istream& istream::operator>> (streambuf* sb)`  
 Extracts characters and stores them in the memory area specified by **sb**.  
 If there is no extracted character, **setstate(failbit)** is called.  
 Return value: `*this`
  
7. `streamsize istream::gcount ( ) const`  
 References **chcount** (number of extracted characters).  
 Return value: `chcount`
  
8. `int_type istream::get ( )`  
 Extracts characters.  
 Return value: If characters are extracted: Extracted characters.  
 If no characters are extracted: Calls **setstate(failbat)**, and `streambuf::eof`.
  
9. `istream& istream::get(char& c)`  
`istream& istream::get(signed char& c)`  
`istream& istream::get(unsigned char& c)`  
  
 Extracts characters and stores them in **c**. If the extracted characters are `streambuf::eop`, **failbit** is specified.  
 Return value: `*this`
  
10. `istream& istream::get (char* s, streamsize n)`  
`istream& istream::get(signed char* s, streamsize n)`  
`istream& istream::get(unsigned char* s, streamsize n)`  
  
 Extracts string literals with size `n-1` and stores them in the memory area specified by **s**. If `ok_ = false` or no characters were extracted, **failbit** is specified.  
 Return value: `*this`

11. `istream& istream::get(char* s, streamsize n, char delim)`  
`istream& istream::get(signed char* s, streamsize n, char delim)`  
`istream& istream::get(unsigned char* s, streamsize n, char delim)`

Extracts string literals with size `n-1` and stores them in the memory area specified by `s`.  
If **delim** is found in the string literal, input is stopped.  
If `ok_ = false` or no characters were extracted, **failbit** is specified.  
Return value: `*this`

12. `istream& istream::get(streambuf& sb)`  
Extracts string literals and stores them in the memory area specified by **sb**.  
If `ok_ = false` or no characters were extracted, **failbit** is specified.  
Return value: `*this`

13. `istream& istream::get(streambuf& sb, char delim)`  
Extracts string literals and stores them in the memory area specified by **sb**.  
If **delim** is found in the string literal, input is stopped.  
If `ok_ = false` or no characters were extracted, **failbit** is specified.  
Return value: `*this`

14. `istream& istream::getline(char* s, streamsize n)`  
`istream& istream::getline(signed char* s, streamsize n)`  
`istream& istream::getline(unsigned char* s, streamsize n)`

Extracts string literals with size `n-1` and stores them in the memory area specified by `s`.  
If `ok_ = false` or no characters were extracted, **failbit** is specified.  
Return value: `*this`

15. `istream& istream::getline(char* s, streamsize n, char delim)`  
`istream& istream::getline(signed char* s, streamsize n, char delim)`  
`istream& istream::getline(unsigned char* s, streamsize n, char delim)`

Extracts string literals with size `n-1` and stores them in the memory area specified by `s`.  
If character **delim** is found, input is stopped.  
If `ok_ = false` or no characters were extracted, **failbit** is specified.  
Return value: `*this`

16. `istream& istream::ignore (streamsize n=1, int_type delim=streambuf::eof)`

Skips reading the number of characters specified by **n**.

If character **delim** is found, skipping is stopped.

Return value: `*this`

17. `int_type istream::peek ( )`

Seeks input characters that can be acquired next.

Return value: If `ok_ = false::streambuf::eof`

If `ok_ != false: rdbuf( )->sgetc( )`

18. `istream& istream::read (char* s, streamsize n)`

`istream& istream::read(signed char* s, streamsize n)`

`istream& istream::read(unsigned char* s, streamsize n)`

If `ok_ != false`, extracts string literals with size **n** and stores them in the memory area specified by **s**. If the number of extracted characters does not match with the number of **n**, **eofbit** is specified.

Return value: `*this`

19. `streamsize istream::readsome (char* s, streamsize n)`

`streamsize istream::readsome(signed char* s, streamsize n)`

`streamsize istream::readsome(unsigned char* s, streamsize n)`

Extracts string literals with size **n** and stores them in the memory area specified by **s**.

If the number of characters exceeds the stream size, only the number of characters equal to the stream size is stored.

Return value: The number of extracted characters

20. `istream& istream::putback (char c)`

Returns characters **c** to the input stream.

If the characters put back are `streambuf::eof`, **badbit** is specified.

Return value: `*this`

21. `istream& istream::unget ( )`  
Returns the input stream pointer by one.  
If the extracted characters are `streambuf::eof`, **badbit** is specified.  
Return value: `*this`
22. `int istream::sync ( )`  
Checks for an input stream.  
This function calls **`streambuf::pubsync( )`**.  
Return value: If there is no input stream: `streambuf::eof`  
If there is an input stream: 0
23. `pos_type istream::tellg ( )`  
Checks for the input stream position.  
This function calls **`streambuf::pubseekoff(0,cur,in)`**.  
Return value: Offset from the beginning of the stream.  
If an input processing error occurs, -1 is returned.
24. `istream& istream::seekg(pos_type pos)`  
Moves the current stream pointer for **`pos`**.  
This function calls **`streambuf::pubseekpos(pos)`**.  
Return value: `*this`
25. `istream& istream::seekg (off_type off, ios_base::seekdir dir)`  
Moves the position to read the input stream by using the method specified by **`dir`**.  
This function calls **`streambuf::pubseekoff(off,dir)`**. If an input processing error is generated, this processing is not performed.  
Return value: `*this`

## (h) istream Class Manipulator

Type	Definition Name	Description
Function	istream& ws(istream& is)	Skips reading space

1. istream& ws(istream& is)  
Skips reading white space.  
Return value: is

## (i) istream Non-Member Function

Type	Definition Name	Description
Function	istream& operator>>(istream& in, char* s)	Extracts character strings and stores them 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 characters and stores them in c
	istream& operator>>(istream& in, signed char& c)	
	istream& operator>>(istream& in, unsigned char& c)	

1. istream& operator>>(istream& in, char\* s)  
istream& operator>>(istream& in, signed char\* s)  
istream& operator>>(istream& in, unsigned char\* s)

Extracts character strings and stores them in the memory area specified by **s**. Processing is terminated when

- the number of characters stored equals field width – 1
- streambuf::eof is found in the input line
- the next input enabled character **c** is isspace(c)=1

If no characters are stored, **failbit** is specified.

Return value: in

2. istream& operator>>(istream& in, char& c)  
istream& operator>>(istream& in, signed char& c)  
istream& operator>>(istream& in, unsigned char& c)

Extracts characters and stores them in **c**. If no characters are stored, **failbit** is specified.

Return value: in



## (j) ostream::sentry Class

### Definition Names

Type	Definition Name	Description
Variable	ok_	Whether the current state is output enabled
	_ec2p_os	Pointer to the ostream object
Function	sentry(ostream& os)	Constructor
	~sentry( )	Destructor
	operator bool( )	References ok_

1. ostream::sentry::sentry (ostream& os)  
Constructor of internal class **sentry**.  
If **good( )** is non-zero and **tie( )** is non-zero, **flush( )** is called.  
Specifies os in \_ec2p\_os.
2. ostream::sentry::~sentry ( )  
Destructor of internal class **sentry**.  
If \_ec2p\_os->flags( ) & ios\_base::unitbuf is true, **flush( )** is called.
3. ostream::sentry::operator bool ( )  
References ok\_.  
Return value: ok\_.

## (k) ostream Class

Type	Definition Name	Description
Function	ostream(streambuf* sbptr)	Constructor.
	virtual ~ostream( )	Destructor.
	ostream & operator<<(bool n)	Inserts n in the output stream.
	ostream & operator<<(short n)	
	ostream & operator<<(unsigned short n)	
	ostream & operator<<(int n)	
	ostream & operator<<(unsigned int n)	
	ostream & operator<<(long n)	
	ostream & operator<<(unsigned long n)	
	ostream & operator<<(float n)	
	ostream & operator<<(double n)	
	ostream & operator<<(long double n)	
	ostream & operator<<(void* n)	
	ostream & operator<<(streambuf* sbptr)	
	ostream & putc(char c)	
		Inserts characters c into the output stream.

Type	Definition Name	Description
Function	ostream & write( const char* s, streamsize n)	Inserts n number of 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::pubseekoff(0,cur,out).
	ostream& seekp(pos_type pos)	Calculates the offset from the beginning of the stream to the current position. Moves the current stream pointer for pos. This function calls streambuf::pubseekpos(pos).
	ostream& seekp(off_type off, seekdir dir)	Moves the stream write position for off, from dir. This function calls streambuf::pubseekoff(off,dir).

1. ostream::ostream (streambuf\* sbptr)  
Constructor.  
Calls **ios (sbptr)**.
2. virtual ostream::~ostream ( )  
Destructor.

3. ostream& ostream::operator<< (bool n)  
 ostream& ostream::operator<< (short n)  
 ostream& ostream::operator<< (unsigned short n)  
 ostream& ostream::operator<< (int n)  
 ostream& ostream::operator<< (unsigned int n)  
 ostream& ostream::operator<< (long n)  
 ostream& ostream::operator<< (unsigned long n)  
 ostream& ostream::operator<< (float n)  
 ostream& ostream::operator<< (double n)  
 ostream& ostream::operator<< (long double n)  
 ostream& ostream::operator<< (void\* n)

If sentry::ok\_ = true, **n** is inserted into the output stream.

If sentry::ok\_ = false, **failbit** is specified.

Return value: \*this

4. ostream& ostream::operator<< (streambuf\* sbptr)  
 If sentry::ok\_ = true, the output string of **sbptr** is inserted into the output stream.  
 If sentry::ok\_ = false, **failbit** is specified.  
 Return value: \*this

5. ostream& ostream::putc (char c)  
 If sentry::ok\_ = true and rdbuf( )->sputc(c)!=streambuf::eof, **c** is inserted into the output stream.  
 Otherwise **failbit** is specified.  
 Return value: \*this

6. ostream& ostream::write (const char\* s, streamsize n)  
 ostream& ostream::write(const signed char\* s, streamsize n)  
 ostream& ostream::write(const unsigned char\* s, streamsize n)

If sentry::ok\_ = true and rdbuf( )->sputn(s, n)=n, **n** number of characters from **s** is inserted to the output stream.

Otherwise **badbit** is specified.

Return value: \*this

7. `ostream& ostream::flush ( )`  
Flushes the output stream.  
This function calls **`streambuf::pubsync( )`**.  
Return value: `*this`
8. `pos_type ostream::tellp ( )`  
Calculates the current write position.  
This function calls **`streambuf::pubseekoff(0,cur,out)`**.  
Return value: The current stream position.  
If an error occurs during processing, -1 is returned.
9. `ostream& ostream::seekp (pos_type pos)`  
If no error occurs, the offset from the beginning of the stream to the current position is calculated.  
Moves the current stream buffer pointer for **`pos`**.  
This function calls **`streambuf::pubseekpos(pos)`**.  
Return value: `*this`
10. `ostream& ostream::seekp (off_type off, seekdir dir)`  
Moves the stream position for **`off`**, from **`dir`**.  
This function calls **`streambuf::pubseekoff(pos,dir)`**.  
Return value: `*this`

## (I) ostream Class Manipulator

Type	Definition Name	Description
Function	ostream& endl(ostream& os)	Adds a new line and flushes the output stream
	ostream& ends(ostream& os)	Adds a NULL code
	ostream& flush(ostream& os)	Flushes the output stream

1. ostream& endl(ostream& os)  
Adds a new line code (end of line indicator) and flushes the output stream.  
This function calls **flush ( )**.  
Return value: os
2. ostream& ends(ostream& os)  
Inserts a NULL code to the output line.  
Return value: os
3. ostream& flush(ostream& os)  
Flushes the output stream.  
This function calls **stream::sync( )**.  
Return value: os

## (m) ostream Non-Member Function

Type	Definition Name	Description
Function	ostream& operator<<(ostream& os, char s)	Inserts s into the output stream
	ostream& operator<<(ostream& os, signed char s)	
	ostream& operator<<(ostream& os, unsigned char s)	
	ostream& operator<<(ostream& os, const char* s)	
	ostream& operator<<(ostream& os, const signed char* s)	
	ostream& operator<<(ostream& os, const unsigned char* s)	

1. ostream& operator<<(ostream& os, char s)  
ostream& operator<<(ostream& os, signed char s)  
ostream& operator<<(ostream& os, unsigned char s)  
ostream& operator<<(ostream& os, const char\* s)  
ostream& operator<<(ostream& os, const signed char\* s)  
ostream& operator<<(ostream& os, const unsigned char\* s)

If `sentry::ok_ = true` and an error does not occur, **s** is inserted into the output stream.  
Otherwise **failbit** is specified.

Return value: os

## (n) smanip Class Manipulator

Type	Definition Name	Description
Function	smanip resetiosflags(ios_base::fmtflags mask)	Clears the flag specified by the mask value
	smanip setiosflags(ios_base::fmtflags mask)	Specifies the format flag (fmtfl)
	smanip setbase(int base)	Sets the radix used at output
	smanip setfill(char c)	Specifies the fill character (fillch)
	smanip setprecision(int n)	Specifies the precision (prec)
	smanip setw(int n)	Specifies the field width (wide)

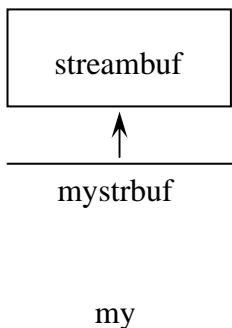
1. smanip resetiosflags(ios\_base::fmtflags mask)  
Clears the flag specified by the mask value.  
Return value: Target object of input/output
2. smanip setiosflags(ios\_base::fmtflags(0), mask)  
Specifies the format flag (**fmtfl**).  
Return value: Target object of input/output
3. smanip setbase(int base)  
Sets the radix used at output.  
Return value: Target object of input/output
4. smanip setfill(char c);  
Specifies the fill characters.  
Return value: Target object of input/output
5. smanip setprecision(int n)  
Specifies the precision.  
Return value: Target object of input/output
6. smanip setw(int n)  
Specifies the field width.  
Return value: Target object of input/output



### (o) Example of Using EC++ Input/Output Libraries

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 objects **istream** and **ostream**.

The following shows the hierarchy of these classes. An arrow (->) indicates that a derived class refers to a base class.



Type	Definition Name	Description
Variable	<code>_file_ptr</code>	File pointer.
Function	<code>mystrbuf( )</code>	Constructor.
	<code>mystrbuf(void* ptr)</code>	Initializes the streambuf buffer.
	<code>virtual ~mystrbuf( )</code>	Destructor.
	<code>void* myfpnr( ) const</code>	Returns a pointer to the FILE type structure.
	<code>mystrbuf* open(const char* filename, int mode)</code>	Specifies the file name and mode and opens file.
	<code>mystrbuf* close( )</code>	Closes file.
	<code>virtual streambuf* setbuf(char* s, streamsize n)</code>	Reserves stream input/output buffer.
	<code>virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode= (ios_base::openmode) (ios_base::in   ios_base::out))</code>	Changes the position of the stream pointer.
	<code>virtual pos_type seekpos(pos_type sp, ios_base::openmode= (ios_base::openmode) (ios_base::in   ios_base::out))</code>	Changes the position of the stream pointer.
	<code>virtual int sync( )</code>	Flushes the stream.
	<code>virtual int showmanyc( )</code>	Returns the number of valid characters of input line.
	<code>virtual int_type underflow( )</code>	Reads one character without moving the stream position.
	<code>virtual int_type pbackfail(int_type c = streambuf::eof)</code>	Puts back the character specified by c.
	<code>virtual int_type overflow(int_type c = streambuf::eof)</code>	Inserts character specified by c.
	<code>void _Init(_f_type* fp)</code>	Initial processing.

## <Example>

```
#include <istream>
#include <ostream>
#include <mystrbuf>
#include <string>
#include <new>
void main(void)
{
    mystrbuf myfin(stdin);
    mystrbuf myfout(stdout);
    istream mycin(&myfin);
    ostream mycout(&myfout);

    int i;
    short s;
    long l;
    char c;
    string str;

    mycin >> i >> s >> l >> c >> str;
    mycout << "This is EC++ Library." << endl
           << i << s << l << c << str << endl;

    return;
}
```

### (3) Memory Management Library

The header file for the memory management library is as follows.

<new>

Defines memory allocation/deallocation function. By setting an exception processing function address to the `_ec2p_new_handler` variable, exception processing can be executed when memory allocation fails. The `_ec2p_new_handler` is a static variable and the initial value is NULL. If this handler is used, reentrant will be lost.

Operations required for the exception processing function:

- Creates an allocatable area and returns the area.
- Operations are not prescribed for cases where an area cannot be created and returned.

Type	Definition Name	Description
Macro	<code>new_handler</code>	Pointer type to the function that returns a void type
Variable	<code>_ec2p_new_handler</code>	Pointer to an exception processing function
Function	<code>void* operator new(size_t size)</code>	Allocates memory area with a size specified by size
	<code>void* operator new[ ](size_t size)</code>	Allocates array area with a size specified by size
	<code>void* operator new(size_t size, void* ptr)</code>	Allocates the area specified by ptr as the memory area
	<code>void* operator new[ ](size_t size, void* ptr)</code>	Allocates the area specified by ptr as the array area
	<code>void operator delete(void* ptr)</code>	Deallocates the memory area
	<code>void operator delete[ ](void* ptr)</code>	Deallocates the array area
	<code>new_handler set_new_handler(new_handler new_P)</code>	Sets exception processing function address (new_P) in <code>_ec2p_new_handler</code>

1. `void* operator new(size_t size)`  
Allocates a memory area with the size specified by **size**.  
If no memory area is allocated and when the **new\_handler** is set, **new\_handler** is called.  
Return value: If memory allocation succeeds: Pointer to void type  
If memory allocation fails: NULL
2. `void* operator new[ ](size_t size)`  
Allocates an array area with the size specified by **size**.  
If no array area is allocated and when the **new\_handler** is set, **new\_handler** is called.  
Return value: If memory allocation succeeds: Pointer to void type  
If memory allocation fails: NULL
3. `void* operator new(size_t size, void* ptr)`  
Allocates the area specified by **ptr** as the memory area.  
Return value: `ptr`
4. `void* operator new[ ](size_t size, void* ptr)`  
Allocates the area specified by **ptr** as the array area.  
Return value: `ptr`
5. `void operator delete(void* ptr)`  
Deallocates the memory area specified by **ptr**.  
If **ptr** is NULL, no operation will be performed.
6. `void operator delete[ ](void* ptr)`  
Deallocates the array area specified by **ptr**.  
If **ptr** is NULL, no operation will be performed.
7. `new_handler set_new_handler(new_handler new_P)`  
Sets `new_P` in `_ec2p_new_handler`.  
Return value: Value of `_ec2p_new_handler`

#### (4) Complex Number Calculation Class Libraries

The header file for the complex number calculation class libraries is as follows.

1. `<complex>`

Defines `float_complex` class and `double_complex` class.

These classes have no hierarchy.

**(a) float\_complex Class**

Type	Definition Name	Description
Type	value_type	float type.
Variable	_re	Defines the real part of float precision.
	_im	Defines the imaginary part of float precision.
Function	float_complex(float re = 0.0f, float im = 0.0f)	Constructor.
	float_complex(const double_complex& rhs)	
	float real( ) const	Calculates the real part (_re).
	float imag( ) const	Calculates the imaginary part (_im).
	float_complex& operator=(float rhs)	Copies rhs to the real part. 0.0f is specified for 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 by rhs and stores the product in *this.
	float_complex& operator/=(float rhs)	Divides 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 and stores the sum in *this.
	float_complex& operator-=(const float_complex& rhs)	Subtracts rhs and stores the difference in *this.
	float_complex& operator*=(const float_complex& rhs)	Multiplies by rhs and stores the product in *this.
	float_complex& operator/=(const float_complex& rhs)	Divides by rhs and stores the quotient in *this.

1. `float_complex::float_complex (float re=0.0f, float im=0.0f)`  
 Constructor of class `float_complex`.  
 The initial settings are as follows:  
`_re = re;`  
`_im = im;`
2. `float_complex::float_complex(const double_complex& rhs)`  
 Constructor of class `float_complex`.  
 The initial settings are as follows:  
`_re = (float)rhs.real( );`  
`_im = (float)rhs.imag( );`
3. `float float_complex::real ( ) const`  
 Calculates the real part.  
 Return value: `this->_re`
4. `float float_complex::imag ( ) const`  
 Calculates the imaginary part.  
 Return value: `this->_im`
5. `float_complex& float_complex::operator= (float rhs)`  
 Copies **rhs** to the real part (**\_re**).  
 0.0f is specified for the imaginary part (**\_im**).  
 Return value: `*this`
6. `float_complex& float_complex::operator+=(float rhs)`  
 Adds **rhs** to the real part (**\_re**) and stores the sum in the real part (**\_re**).  
 The value of the imaginary part (**\_im**) does not change.  
 Return value: `*this`
7. `float_complex& float_complex::operator-= (float rhs)`  
 Subtracts **rhs** from the real part and stores the difference in the real part (**\_re**).  
 The value of the imaginary part (**\_im**) does not change.  
 Return value: `*this`



8. `float_complex& float_complex::operator*= (float rhs)`  
Multiplies by **rhs** and stores the product in `*this`.  
`( _re=_re*rhs, _im=_im*rhs)`  
Return value: `*this`
9. `float_complex& float_complex::operator/= (float rhs)`  
Divides by **rhs** and stores the quotient in `*this`.  
`( _re=_re/rhs, _im=_im/rhs)`  
Return value: `*this`
10. `float_complex& float_complex::operator= (const float_complex& rhs)`  
Copies **rhs**  
Return value: `*this`
11. `float_complex& float_complex::operator+= (const float_complex& rhs)`  
Adds **rhs** and stores the sum in `*this`  
Return value: `*this`
12. `float_complex& float_complex::operator-=(const float_complex& rhs)`  
Subtracts **rhs** and stores the difference in `*this`.  
Return value: `*this`
13. `float_complex& float_complex::operator*= (const float_complex& rhs)`  
Multiplies by **rhs** and stores the product in `*this`.  
Return value: `*this`
14. `float_complex& float_complex::operator/= (const float_complex& rhs)`  
Divides by **rhs** and stores the quotient in `*this`.  
Return value: `*this`

## (b) float\_complex Non-Member Function

Type	Definition Name	Description
Function	float_complex operator+( const float_complex& lhs)	Performs unary + operation of lhs
	float_complex operator+( const float_complex& lhs, const float_complex& rhs)	Adds lhs to rhs and stores the sum in 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)	Performs unary - operation of lhs
	float_complex operator-( const float_complex& lhs, const float_complex& rhs)	Subtracts rhs from lhs and stores the difference in 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)	Multiplies lhs by rhs and stores the product in 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)	Divides lhs by rhs and stores the quotient in lhs
	float_complex operator/ ( const float_complex& lhs, const float& rhs)	

Type	Definition Name	Description
Function	<code>float_complex operator/ (const float&amp; lhs, const float_complex&amp; rhs)</code>	Divides lhs by rhs and stores the quotient in lhs
	<code>bool operator==(const float_complex&amp; lhs, const float_complex&amp; rhs)</code>	Compares the real parts of lhs and rhs, and the imaginary parts of lhs and rhs
	<code>bool operator==(const float_complex&amp; lhs, const float&amp; rhs)</code>	
	<code>bool operator==(const float&amp; lhs, const float_complex&amp; rhs)</code>	
	<code>bool operator!=(const float_complex&amp; lhs, const float_complex&amp; rhs)</code>	Compares the real parts of lhs and rhs, and the imaginary parts of lhs and rhs
	<code>bool operator!=(const float_complex&amp; lhs, const float&amp; rhs)</code>	
	<code>bool operator!=(const float&amp; lhs, const float_complex&amp; rhs)</code>	
	<code>istream&amp; operator&gt;&gt;(istream&amp; is, float_complex&amp; x)</code>	Inputs x in a format of u, (u), or (u,v)(u: real part, v: imaginary part)
	<code>ostream&amp; operator&lt;&lt;(ostream&amp; os, float_complex&amp; x)</code>	Outputs x in a format of u, (u), or (u,v)(u: real part, v: imaginary part)
	<code>float real(const float_complex&amp; x)</code>	Calculates the real part
	<code>float imag(const float_complex&amp; x)</code>	Calculates the imaginary part
	<code>float abs(const float_complex&amp; x)</code>	Calculates the absolute value
	<code>float arg(const float_complex&amp; x)</code>	Calculates the phase angle
	<code>float norm(const float_complex&amp; x)</code>	Calculates the absolute value of the square
	<code>float_complex conj(const float_complex&amp; x)</code>	Calculates the conjugate complex number

Type	Definition Name	Description
Function	<code>float_complex polar(const float&amp; rho, const float&amp; theta)</code>	Calculates the float_complex value for a complex number with size rho and phase angle theta
	<code>float_complex cos(const float_complex&amp; x)</code>	Calculates the complex cosine
	<code>float_complex cosh(const float_complex&amp; x)</code>	Calculates the complex hyperbolic cosine
	<code>float_complex exp(const float_complex&amp; x)</code>	Calculates the exponent function
	<code>float_complex log(const float_complex&amp; x)</code>	Calculates the natural logarithm
	<code>float_complex log10(const float_complex&amp; x)</code>	Calculates the common logarithm
	<code>float_complex pow(const float_complex&amp; x, int y)</code>	Calculates the x to the yth power
	<code>float_complex pow(const float_complex&amp; x, const float&amp; y)</code>	
	<code>float_complex pow(const float_complex&amp; x, const float_complex&amp; y)</code>	
	<code>float_complex pow(const float&amp; x, const float_complex&amp; y)</code>	
	<code>float_complex sin(const float_complex&amp; x)</code>	Calculates the complex sine
	<code>float_complex sinh(const float_complex&amp; x)</code>	Calculates the complex hyperbolic sine
	<code>float_complex sqrt(const float_complex&amp; x)</code>	Calculates the square root within the right half space
	<code>float_complex tan(const float_complex&amp; x)</code>	Calculates the complex tangent
	<code>float_complex tanh(const float_complex&amp; x)</code>	Calculates the complex hyperbolic tangent

1. `float_complex operator+ (const float_complex& lhs)`  
Performs unary + operation of **lhs**.  
Return value: **lhs**
2. `float_complex operator+(const float_complex& lhs, const float_complex& rhs)`  
`float_complex operator+(const float_complex& lhs, const float& rhs)`  
`float_complex operator+(const float& lhs, const float_complex& rhs)`

Adds **lhs** to **rhs** and stores the sum in **lhs**.

Return value: `float_complex(lhs)+=rhs`

3. `float_complex operator-(const float_complex& lhs)`  
Performs unary - operation of **lhs**.  
Return value: `float_complex(-lhs.real( ),-lhs.imag( ))`
4. `float_complex operator-(const float_complex& lhs, const float_complex& rhs)`  
`float_complex operator-(const float_complex& lhs, const float& rhs)`  
`float_complex operator-(const float& lhs, const float_complex& rhs)`

Subtracts **rhs** from **lhs** and stores the difference in **lhs**.

Return value: `float_complex(lhs)-=rhs`

5. `float_complex operator*(const float_complex& lhs, const float_complex& rhs)`  
`float_complex operator*(const float_complex& lhs, const float& rhs)`  
`float_complex operator*(const float& lhs, const float_complex& rhs)`

Multiplies **lhs** by **rhs** and stores the product in **lhs**.

Return value: `float_complex(lhs)*=rhs`

6. `float_complex operator/(const float_complex& lhs, const float_complex& rhs)`  
`float_complex operator/(const float_complex& lhs, const float& rhs)`  
`float_complex operator/(const float& lhs, const float_complex& rhs)`

Divides **lhs** by **rhs** and stores the quotient in **lhs**.

Return value: `float_complex(lhs)/=rhs`

7. `bool operator==(const float_complex& lhs, const float_complex& rhs)`  
`bool operator==(const float_complex& lhs, const float& rhs)`  
`bool operator==(const float& lhs, const float_complex& rhs)`

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.

For a float type parameter, the imaginary part is assumed 0.0f.

Return value: `lhs.real() == rhs.real() && lhs.imag() == rhs.imag()`

8. `bool operator!=(const float_complex& lhs, const float_complex& rhs)`  
`bool operator!=(const float_complex& lhs, const float& rhs)`  
`bool operator!=(const float& lhs, const float_complex& rhs)`

Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.

For a float type parameter, the imaginary part is assumed 0.0f.

Return value: `lhs.real() != rhs.real() || lhs.imag() != rhs.imag()`

9. `istream& operator>>(istream& is, float_complex& x)`  
Inputs **x** in a format of `u`, `(u)`, or `(u,v)` (`u`: real part, `v`: imaginary part).  
The input value is converted to `float_complex`.  
If **x** is input in a format other than the `u`, `(u)`, or `(u,v)` format, `is.setstate(ios_base::failbit)` is called.  
Return value: `is`

10. `ostream& operator<<(ostream& os, const float_complex& x)`  
Outputs **x** to **os**.  
The output format is `u`, `(u)`, or `(u,v)` (`u`: real part, `v`: imaginary part).  
Return value: `os`

11. `float real(const float_complex& x)`  
Calculates the real part.  
Return value: `x.real()`

12. `float imag(const float_complex& x)`  
Calculates the imaginary part.  
Return value: `x.imag()`

13. `float abs(const float_complex& x)`  
Calculates the absolute value.  
Return value: `|x.real()| + |x.imag()|`

14. `float arg(const float_complex& x)`  
Calculates the phase angle.  
Return value: `atan2f(x.imag( ), x.real( ))`
15. `float norm(const float_complex& x)`  
Calculates the absolute value of the square.  
Return value: `x.real( )^2+ x.imag( )^2`
16. `float_complex conj(const float_complex& x)`  
Calculates the conjugate complex number.  
Return value: `float_complex(x.real( ), (-1)*x.imag( ))`
17. `float_complex polar(const float& rho, const float& theta)`  
Calculates the `float_complex` value for a complex number with size **rho** and phase angle (argument) **theta**.  
Return value: `float_complex(rho*cosf(theta), rho*sinf(theta))`
18. `float_complex cos(const float_complex& x)`  
Calculates the complex cosine.  
Return value: `float_complex(cosf(x.real( ))*coshf(x.imag( )), (-1)*sinf(x.real( ))*sinhf(x.imag( )))`
19. `float_complex cosh(const float_complex& x)`  
Calculates the complex hyperbolic cosine.  
Return value: `cos(float_complex((-1)*x.imag( ), x.real( )))`
20. `float_complex exp(const float_complex& x)`  
Calculates the exponential function.  
Return value: `expf(x.real( ))*cosf(x.imag( )),expf(x.real( ))*sinf(x.imag( ))`
21. `float_complex log(const float_complex& x)`  
Calculates the natural logarithm (base e).  
Return value: `float_complex(logf(x)), arg(x))`
22. `float_complex log10(const float_complex& x)`  
Calculates the common logarithm (base 10).  
Return value: `float_complex(log10f(abs(x)), arg(x)/logf(10))`

23. `float_complex pow(const float_complex& x, int y)`  
`float_complex pow(const float_complex& x, const float& y)`  
`float_complex pow(const float_complex& x, const float_complex& y)`  
`float_complex pow(const float& x, const float_complex& y)`  
 Calculates the **x** to the **y**th power.  
 If `pow(0,0)`, a domain error will occur.  
 Return value: For `float_complex pow (const float_complex& x,const float_complex& y)`:  
     `exp(y*logf(x))`  
     Otherwise: `exp(y*log(x))`
24. `float_complex sin(const float_complex& x)`  
 Calculates the complex sine.  
 Return value: `float_complex(sinf(x.real( ))*coshf(x.imag( )), cosf(x.real( ))*sinhf(x.imag( )))`
25. `float_complex sinh (const float_complex& x)`  
 Calculates the complex hyperbolic sine.  
 Return value: `float_complex(0,-1)*sin(float_complex((-1)*x.imag( ),x.real( )))`
26. `float_complex sqrt(const float_complex& x)`  
 Calculates the square root within the right half space.  
 Return value: `float_complex(sqrtf(abs(x))*cosf(arg (x)/2, sqrtf(abs(x))*sinf(arg(x)/2))`
27. `float_complex tan(const float_complex& x)`  
 Calculates the complex tangent.  
 Return value: `sin(x) / cos(x)`
28. `float_complex tanh(const float_complex& x)`  
 Calculates the complex hyperbolic tangent.  
 Return value: `sinh(x) / cosh(x)`



### (c) double\_complex Class

Type	Definition Name	Description
Type	value_type	double type.
Variable	_re	Defines the real part of double precision.
	_im	Defines the imaginary part of double precision.
Function	double_complex( double re=0.0, double im=0.0)	Constructor.
	double_complex(const float_complex&)	
	double real( ) const	Calculates the real part.
	double imag( ) const	Calculates the imaginary part.
	double_complex& operator=(double rhs)	Copies rhs to the real part. 0.0 is specified for the imaginary part.
	double_complex& operator+=(double rhs)	Adds rhs to the real part and stores the sum in *this.
	double_complex& operator-=(double rhs)	Subtracts rhs from the real part and stores the difference in *this.
	double_complex& operator*=(double rhs)	Multiplies by rhs and stores the product in *this.
	double_complex& operator/=(double rhs)	Divides 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 and stores the sum in *this.
	double_complex& operator-=( const double_complex& rhs)	Subtracts rhs and stores the difference in *this.
	double_complex& operator*=( const double_complex& rhs)	Multiplies by rhs and stores the product in *this.
	double_complex& operator/=( const double_complex& rhs)	Divides by rhs and stores the quotient in *this.

1. `double_complex::double_complex(double re=0.0, double im=0.0)`  
 Constructor of class `double_complex`.  
 The initial settings are as follows:  
`_re = re;`  
`_im = im;`
  
2. `double_complex::double_complex(const float_complex&)`  
 Constructor of class `double_complex`.  
 The initial settings are as follows:  
`_re = (double)rhs.real( );`  
`_im = (double)rhs.imag( );`
  
3. `double double_complex::real( ) const`  
 Calculates the real part.  
 Return value: `this->_re`
  
4. `double double_complex::imag( ) const`  
 Calculates the imaginary part.  
 Return value: `this->_im`
  
5. `double_complex& double_complex::operator=(double rhs)`  
 Copies **rhs** to the real part (**\_re**).  
 0.0 is specified for the imaginary part (**\_im**).  
 Return value: `*this`
  
6. `double_complex& double_complex::operator+=(double rhs)`  
 Adds **rhs** to the real part (**\_re**) and stores the sum in the real part (**\_re**).  
 The value of the imaginary part (**\_im**) does not change.  
 Return value: `*this`
  
7. `double_complex& double_complex::operator-=(double rhs)`  
 Subtracts **rhs** from the real part and stores the difference in the real part (**\_re**).  
 The value of the imaginary part (**\_im**) does not change.  
 Return value: `*this`

8. `double_complex& double_complex::operator*= (double rhs)`  
Multiplies by **rhs** and stores the product in `*this`.  
(`_re=_re*rhs, _im=_im*rhs`)  
Return value: `*this`
9. `double_complex& double_complex::operator/= (double rhs)`  
Divides by **rhs** and stores the quotient in `*this`.  
(`_re=_re/rhs, _im=_im/rhs`)  
Return value: `*this`
10. `double_complex& double_complex::operator= (const double_complex& rhs)`  
Copies **rhs**.  
Return value: `*this`
11. `double_complex& double_complex::operator+= (const double_complex& rhs)`  
Adds **rhs** and stores the sum in `*this`.  
Return value: `*this`
12. `double_complex& double_complex::operator-= (const double_complex& rhs)`  
Subtracts **rhs** and stores the difference in `*this`.  
Return value: `*this`
13. `double_complex& double_complex::operator*= (const double_complex& rhs)`  
Multiplies by **rhs** and stores the product in `*this`.  
Return value: `*this`
14. `double_complex& double_complex::operator/= (const double_complex& rhs)`  
Divides by **rhs** and stores the quotient in `*this`.  
Return value: `*this`

#### (d) double\_complex Non-Member Function

Type	Definition Name	Description
Function	<code>double_complex operator+( const double_complex&amp; lhs)</code>	Performs unary + operation of lhs
	<code>double_complex operator+( const double_complex&amp; lhs, const double_complex&amp; rhs)</code>	Adds rhs to lhs and stores the sum in lhs
	<code>double_complex operator+( const double_complex&amp; lhs, const double&amp; rhs)</code>	
	<code>double_complex operator+( const double&amp; lhs, const double_complex&amp; rhs)</code>	
	<code>double_complex operator-( const double_complex&amp; lhs)</code>	Performs unary – operation of lhs
	<code>double_complex operator-( const double_complex&amp; lhs, const double_complex&amp; rhs)</code>	Subtracts rhs from lhs and stores the difference in lhs
	<code>double_complex operator-( const double_complex&amp; lhs, const double&amp; rhs)</code>	
	<code>double_complex operator-( const double&amp; lhs, const double_complex&amp; rhs)</code>	
	<code>double_complex operator*( const double_complex&amp; lhs, const double_complex&amp; rhs)</code>	Multiplies lhs by rhs and stores the product in lhs
	<code>double_complex operator*( const double_complex&amp; lhs, const double&amp; rhs)</code>	
	<code>double_complex operator*( const double&amp; lhs, const double_complex&amp; rhs)</code>	
	<code>double_complex operator/ ( const double_complex&amp; lhs, const double_complex&amp; rhs)</code>	Divides lhs by rhs and stores the quotient in lhs
	<code>double_complex operator/ ( const double_complex&amp; lhs, const double&amp; rhs)</code>	

Type	Definition Name	Description
Function	<code>double_complex operator/ (const double&amp; lhs, const double_complex&amp; rhs)</code>	Divides lhs by rhs and stores the quotient in lhs
	<code>bool operator==(const double_complex&amp; lhs, const double_complex&amp; rhs)</code>	Compares the real part of lhs and rhs, and the imaginary parts of lhs and rhs
	<code>bool operator==(const double_complex&amp; lhs, const double&amp; rhs)</code>	
	<code>bool operator==(const double&amp; lhs, const double_complex&amp; rhs)</code>	
	<code>bool operator!=(const double_complex&amp; lhs, const double_complex&amp; rhs)</code>	Compares the real parts of lhs and rhs, and the imaginary parts of lhs and rhs
	<code>bool operator!=(const double_complex&amp; lhs, const double&amp; rhs)</code>	
	<code>bool operator!=(const double&amp; lhs, const double_complex&amp; rhs)</code>	
	<code>istream&amp; operator&gt;&gt;(istream&amp; is, double_complex&amp; x)</code>	Inputs x in a format of u,(u), or (u,v)(u: real part, v: imaginary part)
	<code>ostream&amp; operator&lt;&lt;(ostream&amp; os, double_complex&amp; x)</code>	Outputs x in a format of u,(u), or (u,v)(u: real part, v: imaginary part)
	<code>double real(const double_complex&amp; x)</code>	Calculates the real part
	<code>double imag(const double_complex&amp; x)</code>	Calculates the imaginary part
	<code>double abs(const double_complex&amp; x)</code>	Calculates the absolute value
	<code>double arg(const double_complex&amp; x)</code>	Calculates the phase angle
	<code>double norm(const double_complex&amp; x)</code>	Calculates the absolute value of the square
	<code>double_complex conj(const double_complex&amp; x)</code>	Calculates the conjugate complex number

Type	Definition Name	Description
Function	<code>double_complex polar( const double&amp; rho, const double&amp; theta)</code>	Calculates the double_complex value for a complex number with size rho and phase angle theta
	<code>double_complex cos( const double_complex&amp; x)</code>	Calculates the complex cosine
	<code>double_complex cosh( const double_complex&amp; x)</code>	Calculates the complex hyperbolic cosine
	<code>double_complex exp( const double_complex&amp;)</code>	Calculates the exponential function
	<code>double_complex log( const double_complex&amp; x)</code>	Calculates the natural logarithm
	<code>double_complex log10( const double_complex&amp; x)</code>	Calculates the common logarithm
	<code>double_complex pow( const double_complex&amp; x, int y)</code>	Calculates the x to the yth power
	<code>double_complex pow( const double_complex&amp; x, const double&amp; y)</code>	
	<code>double_complex pow( const double_complex&amp; x, const double_complex&amp; y)</code>	
	<code>double_complex pow( const double&amp; x, const double_complex&amp; y)</code>	
	<code>double_complex sin( const double_complex&amp; x)</code>	Calculates the complex sine
	<code>double_complex sinh( const double_complex&amp; x)</code>	Calculates the complex hyperbolic sine
	<code>double_complex sqrt( const double_complex&amp; x)</code>	Calculates the square root within the right half space
	<code>double_complex tan( const double_complex&amp; x)</code>	Calculates the complex tangent
	<code>double_complex tanh( const double_complex&amp; x)</code>	Calculates the complex hyperbolic tangent

1. `double_complex operator+(const double_complex& lhs)`  
Performs unary + operation of **lhs**.  
Return value: **lhs**
2. `double_complex operator+(const double_complex& lhs, const double_complex& rhs)`  
`double_complex operator+(const double_complex& lhs, const double& rhs)`  
`double_complex operator+(const double& lhs, const double_complex& rhs)`  
Adds **lhs** to **rhs** and stores the sum in **lhs**.  
Return value: `double_complex(lhs)+=rhs`
3. `double_complex operator-(const double_complex& lhs)`  
Performs unary - operation of **lhs**.  
Return value: `double_complex(-lhs.real( ), -lhs.imag( ))`
4. `double_complex operator-(const double_complex& lhs, const double_complex& rhs)`  
`double_complex operator-(const double_complex& lhs, const double& rhs)`  
`double_complex operator-(const double& lhs, const double_complex& rhs)`  
Subtracts **rhs** from **lhs** and stores the difference in **lhs**.  
Return value: `double_complex(lhs)-=rhs`
5. `double_complex operator*(const double_complex& lhs, const double_complex& rhs)`  
`double_complex operator*(const double_complex& lhs, const double& rhs)`  
`double_complex operator*(const double& lhs, const double_complex& rhs)`  
Multiplies **lhs** by **rhs** and stores the product in **lhs**.  
Return value: `double_complex(lhs)*=rhs`
6. `double_complex operator/(const double_complex& lhs, const double_complex& rhs)`  
`double_complex operator/(const double_complex& lhs, const double& rhs)`  
`double_complex operator/(const double& lhs, const double_complex& rhs)`  
Divides **lhs** by **rhs** and stores the quotient in **lhs**.  
Return value: `double_complex(lhs)/=rhs`

7. `bool operator==(const double_complex& lhs, const double_complex& rhs)`  
`bool operator==(const double_complex& lhs, const double& rhs)`  
`bool operator==(const double& lhs, const double_complex& rhs)`  
 Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.  
 For a double type parameter, the imaginary part is assumed 0.0.  
 Return value: `lhs.real() == rhs.real() && lhs.imag() == rhs.imag()`
  
8. `bool operator!=(const double_complex& lhs, const double_complex& rhs)`  
`bool operator!=(const double_complex& lhs, const double& rhs)`  
`bool operator!=(const double& lhs, const double_complex& rhs)`  
 Compares the real parts of **lhs** and **rhs**, and the imaginary parts of **lhs** and **rhs**.  
 For a double type parameter, the imaginary part is assumed 0.0.  
 Return value: `lhs.real() != rhs.real() || lhs.imag() != rhs.imag()`
  
9. `istream& operator>>(istream& is, double_complex& x)`  
 Inputs **x** with 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,v)` format, `is.setstate(ios_base::failbit)` is called.  
 Return value: `is`
  
10. `ostream& operator<<(ostream& os, const double_complex& x)`  
 Outputs **x** to `os`.  
 The output format is `u`, `(u)`, or `(u,v)` (`u`: real part, `v`: imaginary part).  
 Return value: `os`
  
11. `double real(const double_complex& x)`  
 Calculates the real part.  
 Return value: `x.real()`
  
12. `double imag(const double_complex& x)`  
 Calculates the imaginary part.  
 Return value: `x.imag()`
  
13. `double abs(const double_complex& x)`  
 Calculates the absolute value.  
 Return value: `|x.real()| + |x.imag()|`



14. `double arg(const double_complex& x)`  
Calculates the phase angle.  
Return value: `atan2(x.imag( ), x.real( ))`
15. `double norm(const double_complex& x)`  
Calculates the absolute value of the square.  
Return value: `x.real( )^2+ x.imag( )^2`
16. `double_complex conj(const double_complex& x)`  
Calculates the conjugate complex number.  
Return value: `double_complex(x.real( ), (-1)*x.imag( ))`
17. `double_complex polar(const double& rho, const double& theta)`  
Calculates the `double_complex` value for a complex number with size **rho** and phase angle (argument) **theta**.  
Return value: `double_complex(rho*cos(theta), rho*sin(theta))`
18. `double_complex cos(const double_complex& x)`  
Calculates the complex cosine.  
Return value: `double_complex(cos(x.real( ))*cosh(x.imag( )), (-1)*sin(x.real( ))*sinh(x.imag( )))`
19. `double_complex cosh(const double_complex& x)`  
Calculates the complex hyperbolic cosine.  
Return value: `cos(double_complex((-1)*x.imag( ), x.real( )))`
20. `double_complex exp(const double_complex& x)`  
Calculates the exponent function.  
Return value: `exp(x.real( ))*cos(x.imag( )),exp(x.real( ))*sin(x.imag( ))`
21. `double_complex log(const double_complex& x)`  
Calculates the natural logarithm (base e).  
Return value: `double_complex(log(abs(x)), arg(x))`

22. `double_complex log10(const double_complex& x)`  
 Calculates the common logarithm (base 10).  
 Return value: `double_complex(log10(abs(x)), arg(x)/log(10))`
23. `double_complex pow(const double_complex& x, int y)`  
`double_complex pow(const double_complex& x, const double& y)`  
`double_complex pow(const double_complex& x, const double_complex& y)`  
`double_complex pow(const double& x, const double_complex& y)`  
  
 Calculates the **x** to the **y**th power.  
 If `pow(0,0)`, a domain error will occur.  
 Return value: `exp(y*log(x))`
24. `double_complex sin(const double_complex& x)`  
 Calculates the complex sine  
 Return value: `double_complex(sin(x.real( ))*cosh(x.imag( )), cos(x.real( ))*sinh(x.imag( )))`
25. `double_complex sinh (const double_complex& x)`  
 Calculates the complex hyperbolic sine  
 Return value: `double_complex(0,-1)*sin(double_complex((-1)*x.imag( ),x.real( )))`
26. `double_complex sqrt(const double_complex& x)`  
 Calculates the square root within the right half space  
 Return value: `double_complex(sqrt(abs(x))*cos(arg(x)/2, sqrt(abs(x))*sin(arg(x)/2)`
27. `double_complex tan(const double_complex& x)`  
 Calculates the complex tangent.  
 Return value: `sin(x) / cos(x)`
28. `double_complex tanh(const double_complex& x)`  
 Calculates the complex hyperbolic tangent.  
 Return value: `sinh(x) / cosh(x)`

## (5) String Handling Class Library

The header file for string handling class library is as follows.

### 1. <string>

Defines the string class. This class has no hierarchy.

#### (a) string Class

Type	Definition Name	Description
Type	iterator	char* type
	const_iterator	const char* type
Constant	npos	Maximum string literal length (UNIT_MAX characters)
Variable	s_ptr	Pointer to the memory area where the string literal is stored by the object
	s_len	The length of the string literal stored by the object
	s_res	Size of the defined memory area to store string literal by the object
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)	Assigns str
	string& operator=(char c)	Assigns c
	iterator begin( )	Calculates the start pointer of the string literal
	const_iterator begin( ) const	
	iterator end( )	Calculates the end pointer of the string literal
	const_iterator end( ) const	

Type	Definition Name	Description
Function	size_t size( ) const	Calculates the length of the stored string literal
	size_t length( ) const	
	size_t max_size( ) const	Calculates the size of the defined memory area
	void resize(size_t n, char c)	Changes the string literal length to n that can be stored
	void resize(size_t n)	Changes the string literal length to n that can be stored
	size_t capacity( ) const	Calculates the size of the defined memory area
	void reserve(size_t res_arg = 0)	Performs re-allocation of the memory area
	void clear( )	Clears the stored string literal
	bool empty( ) const	Checks whether the stored string literal 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)	
	string& operator+=(const string& str)	Adds the string literal stored by str to the object
	string& operator+=(const char* str)	Adds the string literal stored by str to the object
	string& operator+=(char c)	Adds the characters stored by c to the object
	string& append(const string& str)	Adds the string literal stored by str to the object
	string& append(const char* str)	
	string& append( const string& str, size_t pos, size_t n)	Adds n number of characters of the str string literal to the object position pos

Type	Definition Name	Description
Function	string& append(const char* str, size_t n)	Adds n number of characters of the str string literal
	string& append(size_t n, char c)	Adds n number of characters c
	string& assign(const string& str)	Assigns str string literal
	string& assign(const char* str)	
	string& assign( const string& str, size_t pos, size_t n)	Adds n number of characters of the str string literal to position pos
	string& assign(const char* str, size_t n)	Assigns n number of characters of str string literal
	string& assign(size_t n, char c)	Assigns n number of characters c
	string& insert(size_t pos1, const string& str)	Inserts str string literal to position pos1
	string& insert( size_t pos1, const string& str, size_t pos2, size_t n)	Inserts n number of characters to position pos1 from position pos2 of str string literal
	string& insert( size_t pos, const char* str, size_t n)	Inserts n number of characters of string literal str to position pos
	string& insert(size_t pos, const char* str)	Inserts string literal str to position pos
	string& insert(size_t pos, size_t n, char c)	Inserts a string literal of n number of characters c to position pos
	iterator insert(iterator p, char c=char( ))	Inserts characters c at the head of the string literal specified by p

Type	Definition Name	Description
Function	void insert(iterator p, size_t n, char c)	Inserts n number of characters c before the characters specified by p
	string& erase(size_t pos=0, size_t n=npos)	Deletes n number of characters from position pos
	iterator erase(iterator position)	Deletes the characters referenced by position
	iterator erase(iterator first, iterator last)	Deletes the characters in range [first, last]
	string& replace( size_t pos1, size_t n1, const string& str)	Replaces string literal of n1 characters from position pos1 with the str string literal
	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 string literal of n1 characters from position pos1 with string literal of n2 characters from str position pos2
	string& replace( size_t pos, size_t n1, const char* str, size_t n2)	Replaces string literal of n1 characters from position pos with n2 characters of the str string literal
	string& replace( size_t pos, size_t n1, size_t n2, char c)	Replaces string literal of n1 characters from position pos with n2 characters c
	string& replace( iterator i1, iterator i2, const string& str)	Replaces string literal i1 to i2 with the str string literal

Type	Definition Name	Description
Function	string& replace( iterator i1, iterator i2, const char* str)	Replaces string literal i1 to i2 with the str string literal
	string& replace( iterator i1, iterator i2, const char* str, size_t n)	Replaces string literal i1 to i2 with n number of characters of str string literal
	string& replace( iterator i1, iterator i2, size_t n, char c)	Replaces string literal from position i1 to i2 with n number of characters c
	size_t copy( char* str, size_t n, size_t pos=0) const	Copies n characters of string literal str to position pos
	void swap(string& str)	Swaps with str string literal
	const char* c_str( ) const	References the pointer to the memory area where the string literal is stored
	const char* data( ) const	
	size_t find( const string& str, size_t pos=0) const	Finds the position where the string literal same as the str string literal 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 literal same as the n characters of str first appears after position pos
	size_t find(char c, size_t pos=0) const	Finds the position where character c first appears after position pos
	size_t rfind( const string& str, size_t pos=npos) const	Finds the position where a string literal same as the str string literal appears most recently before position pos
	size_t rfind( const char* str, size_t pos=npos) const	

Type	Definition Name	Description
Function	size_t rfind( const char* str, size_t pos, size_t n) const	Finds the position where the string literal 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 the string literal 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 literal 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 the string literal 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 the n characters of string literal 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
	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 the str first appears after position pos
	size_t find_first_not_of( const char* str, size_t pos=0) const	



Type	Definition Name	Description
Function	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 from the start of str to n characters first 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 the 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 from the start of str to n characters appears most recently before position pos.
	size_t find_last_not_of( char c, size_t pos=npos) const	Finds the location 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 with a string literal range [pos,n] for the stored string literal
	int compare(const string& str) const	Compares a string literal with str string literal
	int compare( size_t pos1, size_t n1, const string& str) const	Compares a string literal of n1 characters from position pos1 with str
	int compare( size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const	Compares a string literal of n1 characters from position pos1 with the string literal of n2 characters from str position pos2
	int compare(const char* str) const	Compares with str
	int compare( size_t pos1, size_t n1, const char* str, size_t n2=npos) const	Compares a string literal of n1 characters from position pos1 with n2 characters of str

1. `string::string(void)`  
The settings are as follows:  
`s_ptr = 0;`  
`s_len = 0;`  
`s_res = 0;`
2. `string::string(const string& str, size_t pos=0, size_t n=npos)`  
Copies **str**. Note that **s\_len** will be the smaller value of **n** and **s\_len**.
3. `string::string(const char* str, size_t n)`  
The settings are as follows:  
`s_ptr = str;`  
`s_len = n;`  
`s_res = n+1;`
4. `string::string(const char* str)`  
The settings are as follows:  
`s_ptr = str;`  
`s_len = String literal length of str;`  
`s_res = String literal length of str +1;`
5. `string::string (size_t n, char c)`  
The settings are as follows:  
`s_ptr=String literal of n number of characters c`  
`s_len = n;`  
`s_res = n+1;`
6. `string::~~string( )`  
Destructor of class `string`.  
Deallocates the memory area where the string literal is stored.
7. `string& string::operator= (const string& str)`  
Assigns the **str** data.  
Return value: `*this`

8. `string& string::operator= (const char* str)`  
Creates a string object from **str** and assigns the data of **str** to the string object.  
Return value: `*this`
9. `string& string::operator=(char c)`  
Creates a string object from **c** and assigns the data of **c** to the string object.  
Return value: `*this`
10. `string::iterator string::begin ( )`  
`string::const_iterator string::begin( ) const`  
Calculates the start pointer of the string literal.  
Return value: Start pointer of string literal
11. `string::iterator string::end( )`  
`string::const_iterator string::end( ) const`  
Calculates the end pointer of the string literal.  
Return value: End pointer of string literal
12. `size_t string::size ( ) const`  
`size_t string::length ( ) const`  
Calculates the length of the stored string literal.  
Return value: Length of the stored string literal
13. `size_t string::max_size ( ) const`  
Calculates the size of the defined memory area.  
Return value: Size of the defined area
14. `void string::resize (size_t n, char c)`  
Changes the string literal length to **n** that can be stored.  
If `n<=size( )`, replaces the string literal with the original string literal with length **n**.  
If `n>size( )`, replaces the string literal with a string literal that has **c** added to the end so that the length equal to **n**.  
The length must be `n<=max_size`.  
If `n>max_size( )`, the string literal length is assumed `n=max_size( )`.

15. `void string::resize (size_t n)`  
Changes the string literal length to **n** that can be stored.  
If `n<=size( )`, replaces the string literal with the original string literal with length **n**.  
The length must be `n<=max_size`.
16. `size_t string::capacity ( ) const`  
Calculates the size of the defined memory area.  
Return value: Size of the defined memory area
17. `void string::reserve (size_t res_arg = 0)`  
Re-allocates the memory area.  
After **reserve( )**, **capacity( )** will be equal to or larger than the **reserve( )** parameter.  
When memory area is re-allocated, all references, pointers, and iterator that references the elements of the numeric literal (number sequence, series) become invalid.
18. `void string::clear ( )`  
Clears the stored string literal.
19. `bool string::empty ( ) const`  
Checks whether the stored string literal length is 0.  
Return value: If the length of the stored string literal is 0: true  
If the length of the stored string literal is not 0: false
20. `const char& string::operator[ ] (size_t pos) const`  
`char& string::operator[ ] (size_t pos)`  
`const char& string::at(size_t pos) const`  
`char& string::at (size_t pos)`  
References `s_ptr[pos]`.  
Return value: If `n< s_len`: `s_ptr [pos]`  
If `n>= s_len`: `'\0'`
21. `string& string::operator+= (const string& str)`  
Adds the string literal stored by **str**.  
Return value: `*this`
22. `string& string::operator+= (const char* str)`  
Creates a string object from **str** and adds the string literal to the object.  
Return value: `*this`

23. `string& string::operator+=(char c)`  
Creates a string object from **c** and adds the string literal to the object.  
Return value: `*this`
24. `string& string::append (const string& str)`  
`string& string::append(const char* str)`  
Adds **str** string literal to the object.  
Return value: `*this`
25. `string& string::append(const string& str, size_t pos, size_t n)`  
Adds **n** number of characters of the **str** string literal to the object position **pos**.  
Return value: `*this`
26. `string& string::append(const char* str, size_t n)`  
Adds **n** number of characters of the **str** string literal.  
Return value: `*this`
27. `string& string::append(size_t n, char c)`  
Adds **n** number of characters **c**.  
Return value: `*this`
28. `string& string::assign (const string& str)`  
`string& string::assign(const char* str)`  
Assigns **str** string literal.  
Return value: `*this`
29. `string& string::assign(const string& str, size_t pos, size_t n)`  
Assigns **n** number of characters of **str** string literal to position **pos**.  
Return value: `*this`
30. `string& string::assign (const char* str, size_t n)`  
Assigns **n** number of characters of string literal **str**.  
Return value: `*this`

31. `string& string::assign (size_t n, char c)`  
Assigns **n** number of characters **c**.  
Return value: `*this`
32. `string& string::insert (size_t pos1, const string& str)`  
Inserts **str** string literal to position **pos1**.  
Return value: `*this`
33. `string& string::insert(size_t pos1, const string& str, size_t pos2, size_t n)`  
Inserts **n** number of characters to position **pos1** from **str** string literal position **pos2**.  
Return value: `*this`
34. `string& string::insert(size_t pos, const char* str, size_t n)`  
Inserts **n** number of characters of **str** string literal to position **pos**.  
Return value: `*this`
35. `string& string::insert(size_t pos, const char* str)`  
Inserts string literal **str** to position **pos**.  
Return value: `*this`
36. `string& string::insert (size_t pos, size_t n, char c)`  
Inserts a string literal of **n** number of characters **c** to position **pos**.  
Return value: `*this`
37. `string::iterator string::insert(iterator p, char c=char( ))`  
Inserts character **c** at the head of the string literal specified by **p**.  
Return value: `*this`
38. `void string::insert (iterator p, size_t n, char c)`  
Inserts **n** number of characters **c** before the characters specified by **p**.  
Return value: `*this`
39. `string& string::erase (size_t pos=0, size_t n=npos)`  
Deletes **n** number of characters from position **pos**.  
Return value: `*this`

40. `iterator string::erase (iterator position)`  
Deletes the characters referenced by position.  
Return value: If an iterator exists after the delete elements: The next iterator of deleted elements  
If an iterator does not exist after the deleted elements: `end()`
41. `iterator string::erase(iterator first, iterator last)`  
Deletes the characters in range [**first**, **last**].  
Return value: If an iterator exists after **last**: Iterator after **last**  
If an iterator does not exist after **last**: `\0`
42. `string& string::replace (size_t pos1, size_t n1, const string& str)`  
`string& string::replace(size_t pos1, size_t n1, const char* str)`  
Replaces string literal of **n1** characters from position **pos1** with the **str** string literal.  
Return value: `*this`
43. `string& string::replace(size_t pos, size_t n1, const string& str, size_t pos2, size_t n2)`  
Replaces string literal of **n1** characters from position **pos1** with string literal of **n2** characters from **str** position **pos2**.  
Return value: `*this`
44. `string& string::replace(size_t pos, size_t n1, const char* str, size_t n2)`  
Replaces string literal of **n1** characters from position **pos** with the **str** string literal of **n2** characters  
Return value: `*this`
45. `string& string::replace(size_t pos, size_t n1, size_t n2, char c)`  
Replaces string literal of **n1** characters from position **pos** with **n2** characters **c**.  
Return value: `*this`
46. `string& string::replace(iterator i1, iterator i2, const string& str)`  
`string& string::replace(iterator i1, iterator i2, const char* str)`  
Replaces string literal **i1** to **i2** with the **str** string literal.  
Return value: `*this`
47. `string& string::replace(iterator i1, iterator i2, const char* str, size_t n)`  
Replaces string literal **i1** to **i2** with **n** characters of **str** string literal  
Return value: `*this`

48. `string& string::replace (iterator i1, iterator i2, size_t n, char c)`  
Replaces characters from position **i1** to **i2** with **n** number of characters **c**.  
Return value: `*this`
49. `size_t string::copy (char* str, size_t n, size_t pos=0) const`  
Copies **n** characters of string literal **str** to position **pos**.  
Return value: `rlen`
50. `void string::swap (string& str)`  
Swaps with **str** string literal.
51. `const char* string::c_str ( ) const`  
`const char* string::data ( ) const`  
References the pointer to the area where the string literal is stored.  
Return value: `s_ptr`
52. `size_t string::find(const string& str, size_t pos=0) const`  
`size_t string::find (const char* str, size_t pos=0) const`  
Finds the position where the string literal same as the **str** string literal first appears after position **pos**.  
Return value: Offset of string literal
53. `size_t string::find(const char* str, size_t pos, size_t n) const`  
Finds the position where the string literal same as **n** characters of **str** first appears after position **pos**.  
Return value: Offset of string literal
54. `size_t string::find (char c, size_t pos=0) const`  
Finds the position where character **c** first appears after position **pos**.  
Return value: Offset of string literal
55. `size_t string::rfind (const string& str, size_t pos=npos) const`  
`size_t string::rfind(char *str, size_t pos=npos) const`  
Finds the position where the string literal same as the **str** string literal appears most recently before position **pos**.  
Return value: Offset of string literal



56. `size_t string::rfind(const char* str, size_t pos, size_t n) const`  
Finds the position where the string literal same as **n** characters of **str** appears most recently before position **pos**.  
Return value: Offset of string literal
57. `size_t string::rfind(char c, size_t pos=npos) const`  
Finds the position where character **c** appears most recently before position **pos**.  
Return value: Offset of string literal
58. `size_t string::find_first_of (const string& str, size_t pos=0) const`  
`size_t string::find_first_of(const char* str, size_t pos=0) const`  
Finds the position where any character included in the string literal **str** first appears after position **pos**.  
Return value: Offset of string literal
59. `size_t string::find_first_of(const char* str, size_t pos, size_t n) const`  
Finds the position where any character included in **n** characters of string literal **str** first appears after position **pos**.  
Return value: Offset of string literal
60. `size_t string::find_first_of(char c, size_t pos=0) const`  
Finds the position where character **c** first appears after position **pos**.  
Return value: Offset of string literal
61. `size_t string::find_last_of (const string& str, size_t pos=npos) const`  
`size_t string::find_last_of(const char* str, size_t pos=npos) const`  
Finds the position where any character included in the string literal **str** appears most recently before position **pos**.  
Return value: Offset of string literal
62. `size_t string::find_last_of(const char* str, size_t pos, size_t n) const`  
Finds the position where any character included in **n** characters of string literal **str** appears most recently before position **pos**.  
Return value: Offset of string literal
63. `size_t string::find_last_of(char c, size_t pos=npos) const`  
Finds the position where character **c** appears most recently before position **pos**.  
Return value: Offset of string literal

64. `size_t string::find_first_not_of (const string& str, size_t pos=0) const`  
`size_t string::find_first_not_of(const char* str, size_t pos=0) const`  
Finds the position where a character different from any character included in the **str** first appears after position **pos**.  
Return value: Offset of string literal
65. `size_t string::find_first_not_of(const char* str, size_t pos, size_t n) const`  
Finds the position where a character different from any character from the start of **str** for **n** characters first appears after position **pos**.  
Return value: Offset of string literal
66. `size_t string::find_first_not_of (char c, size_t pos=0) const`  
Finds the position where a character different from character **c** first appears after position **pos**  
Return value: Offset of string literal
67. `size_t string::find_last_not_of (const string& str, size_t pos=npos) const`  
`size_t string::find_last_not_of(const char* str, size_t pos=npos) const`  
Finds the position where a character different from any character included in the **str** appears most recently before position **pos**.  
Return value: Offset of string literal
68. `size_t string::find_last_not_of(const char* str, size_t pos, size_t n) const`  
Finds the position where a character different from any character from the start of **str** to **n** characters appears most recently before position **pos**.  
Return value: Offset of string literal
69. `size_t string::find_last_not_of(char c, size_t pos=npos) const`  
Finds the location where a character different from character **c** appears most recently before position **pos**.  
Return value: Offset of string literal
70. `string string::substr (size_t pos=0, size_t n=npos) const`  
Creates an object with a string literal range [**pos,n**] for the stored string literal.  
Return value: Object address with string literal range [**pos,n**]

71. `int string::compare (const string& str) const`  
Compares a string literal with **str** string literal.  
Return value: If the string literals are the same: 0  
                  If the string literals are different: 1 when `this->s_len > str.s_len`,  
                  -1 when `this->s_len < str.s_len`
72. `int string::compare (size_t pos1, size_t n1, const string& str) const`  
Compares a string literal of **n1** characters from position **pos1** with **str**.  
Return value: If the string literals are the same: 0  
                  If the string literals are different: 1 when `this->s_len > str.s_len`,  
                  -1 when `this->s_len < str.s_len`
73. `int string::compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const`  
Compares a string literal of **n1** characters from position **pos1** with the string literal of **n2** characters from **str** position **pos2**.  
Return value: If the string literals are the same: 0  
                  If the string literals are different: 1 when `this->s_len > str.s_len`,  
                  -1 when `this->s_len < str.s_len`
74. `int string::compare(const char* str) const`  
Compares with **str**.  
Return value: If the string literals are the same: 0  
                  If the string literals are different: 1 when `this->s_len > str.s_len`,  
                  -1 when `this->s_len < str.s_len`
75. `int string::compare(size_t pos1, size_t n1, const char* str, size_t n2=npos) const`  
Compares a string literal of **n1** characters from position **pos1** with **sn2** characters of **str**.  
Return value: If the string literals are the same: 0  
                  If the string literals are different: 1 when `this->s_len > str.s_len`,  
                  -1 when `this->s_len < str.s_len`

## (b) string Class Manipulators

Type	Definition Name	Description
Function	string operator +(const string& lhs, const string& rhs)	Adds the string literal (or character) of rhs to the string literal (or character) of lhs, creates an object and stores the string literal in the object
	string operator +(const char* lhs, const string& rhs)	
	string operator +(char lhs, const string& rhs)	
	string operator +(const string& lhs, const char* rhs)	
	string operator +(const string& lhs, char rhs)	
	bool operator ==(const string& lhs, const string& rhs)	Compares the string literal of lhs with string literal 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 literal of lhs with string literal 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 literal length of lhs with the string literal length of rhs
	bool operator <(const char* lhs, const string& rhs)	
	bool operator <(const string& lhs, const char* rhs)	
	bool operator >(const string& lhs, const string& rhs)	Compares the string literal length of lhs with the string literal length of rhs
	bool operator >(const char* lhs, const string& rhs)	
	bool operator >(const string& lhs, const char* rhs)	

Type	Definition Name	Description
Function	bool operator <=( const string& lhs, const string& rhs)	Compares the string literal length of lhs with the string literal length of rhs
	bool operator <=(const char* lhs, const string& rhs)	
	bool operator <=(const string& lhs, const char* rhs)	
	bool operator >=(const string& lhs, const string& rhs)	Compares the string literal length of lhs with the string literal stored in 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 literal of lhs with the string literal of rhs
	istream& operator >> (istream& is, string& str)	Extracts a string literal in str
	ostream& operator << ( ostream& os, const string& str)	Inserts a string literal
	istream& getline( istream& is, string& str, char delim)	Extracts a string literal from is and adds it to str. If 'delim' is detected, terminates input.
	istream& getline (istream& is, string& str)	Extracts a string literal from is and adds it to str. If a new-line character is detected, terminates input.

1. string operator+(const string& lhs, const string& rhs)  
string operator+(const char\* lhs, const string& rhs)  
string operator+(char lhs, const string& rhs)  
string operator+(const string& lhs, const char\* rhs)  
string operator+(const string& lhs, char rhs)  
Links the string literal (characters) of **lhs** with the strings literal (characters) of **rhs**, creates an object and stores the string literal in the object.  
Return value: Object where the linked string literal is stored
  
2. bool operator==(const string& lhs, const string& rhs)  
bool operator==(const char\* lhs, const string& rhs)  
bool operator==(const string& lhs, const char\* rhs)  
Compares the string literal of **lhs** with the string literal of **rhs**.  
Return value: If the string literals are the same: true  
If the string literals are different: false
  
3. bool operator!=(const string& lhs, const string& rhs)  
bool operator!=(const char\* lhs, const string& rhs)  
bool operator!=(const string& lhs, const char\* rhs)  
Compares the string literal of **lhs** with the string literal of **rhs**.  
Return value: If the string literals are the same: true  
If the string literals are different: false
  
4. bool operator<(const string& lhs, const string& rhs)  
bool operator<(const char\* lhs, const string& rhs)  
bool operator<(const string& lhs, const char\* rhs)  
Compares the string literal length of **lhs** with the string literal length of **rhs**.  
Return value: If lhs.s\_len < rhs.s\_len: true  
If lhs.s\_len >= rhs.s\_len: false
  
5. bool operator>(const string& lhs, const string& rhs)  
bool operator>(const char\* lhs, const string& rhs)  
bool operator>(const string& lhs, const char\* rhs)  
Compares the string literal length of **lhs** with the string literal length of **rhs**.  
Return value: If lhs.s\_len > rhs.s\_len: true  
If lhs.s\_len <= rhs.s\_len: false

6. `bool operator<=(const string& lhs, const string& rhs)`  
`bool operator<=(const char* lhs, const string& rhs)`  
`bool operator<=(const string& lhs, const char* rhs)`  
 Compares the string literal length of **lhs** with the string literal length of **rhs**.  
 Return value: If `lhs.s_len <= rhs.s_len`: true  
                   If `lhs.s_len > rhs.s_len`: false
  
7. `bool operator>=(const string& lhs, const string& rhs)`  
`bool operator>=(const char* lhs, const string& rhs)`  
`bool operator>=(const string& lhs, const char* rhs)`  
 Compares the string literal length of **lhs** with the string literal stored in **rhs**.  
 Return value: If `lhs.s_len >= rhs.s_len`: true  
                   If `lhs.s_len < rhs.s_len`: false
  
8. `void swap(string& lhs, string& rhs)`  
 Swaps the string literal of **lhs** with the string literal of **rhs**.
  
9. `istream& operator>> (istream& is, string& str)`  
 Extracts a string literal in **str**.  
 Return value: `is`
  
10. `ostream& operator<< ostream& os, const string& str)`  
 Inserts a string literal.  
 Return value: `os`
  
11. `istream& getline(istream& is, string& str, char delim)`  
`istream& getline(istream& is, string& str)`  
 Extracts a string literal from **is** and adds it to **str**.  
 If **delim** is detected, terminates input.  
 Return value: `is`
  
12. `istream& getline (istream& is, string& str)`  
 Extracts a string literal from **is** and adds it to **str**.  
 If a new-line character is detected, terminates input.  
 Return value: `is`

### 10.3.3 Reentrant Library

Table 10.42 lists reentrant libraries. The functions that are marked with  $\Delta$  in the table set the **errno** variables. Therefore, the functions can be executed in reentrant unless the program refers to **errno**.

If you want more reentrant capability using a semaphore, specify the **reent** option to the standard library generator. The library then generated is reentrant except for the rand and srand functions. Also note that the behavior of subsequent calls of the strtok function using the same string is not guaranteed. Refer to section, 9.2.2 (7) (b) Specifications of low-level interface routines, and section, 9.2.2 (7) (d) Example of low-level interface routines for reentrant library

**Table 10.42 Reentrant Library List**

No.	Standard Include File		Function Name	Reentrant
1	stddef.h	1	offsetof	O
2	assert.h	2	assert	X
3	ctype.h	3	isalnum	O
		4	isalpha	O
		5	isctrl	O
		6	isdigit	O
		7	isgraph	O
		8	islower	O
		9	isprint	O
		10	ispunct	O
		11	isspace	O
		12	isupper	O
		13	isxdigit	O
		14	tolower	O
		15	toupper	O
4	math.h	16	acos	$\Delta$
		17	asin	$\Delta$
		18	atan	$\Delta$



**Table 10.42 Reentrant Library List (cont)**

No.	Standard Include File		Function Name	Reentrant
4	math.h(cont)	19	atan2	Δ
		20	cos	Δ
		21	sin	Δ
		22	tan	Δ
		23	cosh	Δ
		24	sinh	Δ
		25	tanh	Δ
		26	exp	Δ
		27	frexp	Δ
		28	ldexp	Δ
		31	modf	Δ
		32	pow	Δ
		33	sqrt	Δ
		34	ceil	Δ
		35	fabs	Δ
		36	floor	Δ
		37	fmod	Δ
5	mathf.h	38	acosf	Δ
		39	asinf	Δ
		40	atanf	Δ
		41	atan2f	Δ
		42	cosf	Δ
		43	sinf	Δ
		44	tanf	Δ
		45	coshf	Δ
		46	sinhf	Δ
		47	tanhf	Δ
		48	expf	Δ
		49	frexpf	Δ
		50	ldexpf	Δ
		51	logf	Δ

**Table 10.42 Reentrant Library List (cont)**

<b>No.</b>	<b>Standard Include File</b>		<b>Function Name</b>	<b>Reentrant</b>
5	mathf.h(cont)	52	log10f	Δ
		53	modff	Δ
		54	powf	Δ
		55	sqrtr	Δ
		56	ceilf	Δ
		57	fabsf	Δ
		58	floorf	Δ
		59	fmodf	Δ
6	setjmp.h	60	setjmp	O
		61	longjmp	O
7	stdarg.h	62	va_start	O
		63	va_arg	O
		64	va_end	O
8	stdio.h	65	fclose	X
		66	fflush	X
		67	fopen	X
		68	freopen	X
		69	setbuf	X
		70	setvbuf	X
		71	fprintf	X
		72	fscanf	X
		73	printf	X
		74	scanf	X
		75	sprintf	Δ
		76	sscanf	Δ
		77	vfprintf	X
		78	vprintf	X
		79	vsprintf	Δ
		80	fgetc	X
		81	fgets	X
		82	fputc	X
		83	fputs	X

**Table 10.42 Reentrant Library List (cont)**

<b>No.</b>	<b>Standard Include File</b>		<b>Function Name</b>	<b>Reentrant</b>
8	stdio.h (cont)	84	getc	X
		85	getchar	X
		86	gets	X
		87	putc	X
		88	putchar	X
		89	puts	X
		90	ungetc	X
		91	fread	X
		92	fwrite	X
		93	fseek	X
		94	ftell	X
		95	rewind	X
		96	clearerr	X
		97	feof	X
		98	ferror	X
		99	perror	X
9	stdlib.h	100	atof	Δ
		101	atoi	Δ
		102	atol	Δ
		103	strtod	Δ
		104	strtol	Δ
		105	rand	X
		106	srand	X
		107	calloc	X
		108	free	X
		109	malloc	X
		110	realloc	X
		111	bsearch	O
		112	qsort	O
		113	abs	O
		114	div	Δ
		115	labs	O
		116	ldiv	Δ

**Table 10.42 Reentrant Library List (cont)**

No.	Standard Include File		Function Name	Reentrant
10	string.h	117	memcpy	O
		118	strcpy	O
		119	strncpy	O
		120	strcat	O
		121	strncat	O
		122	memcmp	O
		123	strcmp	O
		124	strncmp	O
		125	memchr	O
		126	strchr	O
		127	strcspn	O
		128	strpbrk	O
		129	strrchr	O
		130	strspn	O
		131	strstr	O
		132	strtok	X
		133	memset	O
		134	strerror	O
		135	strlen	O
		136	memmove	O

Reentrant column:      O: Reentrant  
                              X: Non-reentrant  
                              Δ: \_errno is set.

### 10.3.4 Unsupported Libraries

Table 10.43 lists the libraries not supported by this compiler.

**Table 10.43 Unsupported Libraries**

No.	Standard Include File	Reentrant
1	locale.h*	setlocale, localeconv
2	signal.h*	signal, raise
3	stdio.h	remove, rename, tmpfile, tmpnam, fgetpos, fsetpos
4	stdlib.h	strtoul, abort, atexit, exit, getenv, system, mblen, mbtowc, wctomb, mbstowcs, wcstombs
5	string.h	strcoll, strxfrm
6	time.h	clock, difftime, mktime, time, asctime, ctime, gmtime, localtime, strftime

Note: The header file is not supported.



## 11.1 Program Elements

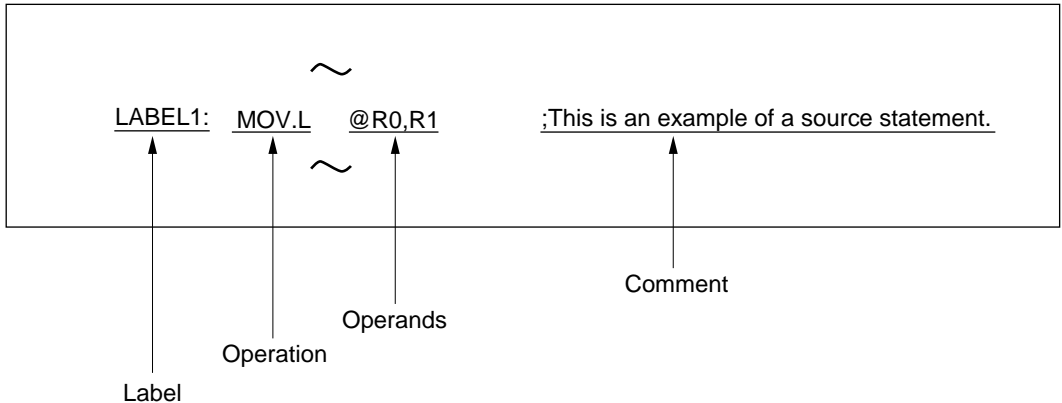
### 11.1.1 Source Statements

#### (1) Source Statement Structure

The following shows the structure of a source statement.

[<label>] [ $\Delta$ <operation>[ $\Delta$ <operand (s)>]] [<comment>]

Example:



#### (a) Label

A symbol or a local symbol is written as a tag attached to a source statement.

A symbol is a name defined by the programmer.

#### (b) Operation

The mnemonic of an executable instruction, an assembler directive, or a preprocessor directive is written as the operation.

Executable instructions are microprocessor instructions.

Assembler directives are instructions that give directions to the assembler.

Preprocessor directives are used for file inclusion, conditional assembly, structured assembly syntax, and macro functions.

(c) Operand

The object(s) of the operation's execution are written as the operand.

The number of operands and their types are determined by the operation. There are also operations which do not require any operands.

(d) Comment

Notes or explanations that make the program easier to understand are written as the comment.

(2) Coding of Source Statements

Source statements are written using ASCII characters. Strings literal and comments can include Japanese characters (shift JIS code or EUC code) or LATIN1 code character.

In principle, a single statement should be written on a single line. The maximum length of a line is 8,192 characters.

(a) Coding of Label

The label is written as follows:

- Written starting in the first column,

Or:

- Written with a colon (:) appended to the end of the label.

Examples:

**LABEL1** ; This label is written starting in the first column.

**LABEL2:** ; This label is terminated with a colon.

---

**LABEL3** ; This label is regarded as an error by the assembler,  
; since it is neither written starting in the first column  
; nor terminated with a colon.



## (b) Coding of Operation

The operation is written as follows:

— When there is no label:

Written starting in the second or later column.

— When there is a label:

Written after the label, separated by one or more spaces or tabs.

Example:

**ADD** R0,R1 ; An example with no label.

**LABEL1: ADD** R1,R2 ; An example with a label.

## (c) Coding of Operand

The operand is written following the operation field, separated by one or more spaces or tabs.

Example:

**ADD R0,R1** ; The ADD instruction takes two operands.

**SHAL R1** ; The SHAL instruction takes one operand.

## (d) Coding of Comment

The comment is written following a semicolon (;).

The assembler regards all characters from the semicolon to the end of the line as the comment.

Example:

**ADD R0,R1 ; Adds R0 to R1.**

### (3) Coding of Source Statements across Multiple Lines

A single source statement can be written across several lines in the following situations:

- When the source statement is too long as a single statement.
- When it is desirable to attach a comment to each operand.

Write source statements across multiple lines using the following procedure.

- (a) Insert a new line after a comma that separates operands.
- (b) Insert a plus sign (+) in the first column of the new line.
- (c) Continue writing the source statement following the plus sign.

Spaces and tabs can be inserted following the plus sign. A comment can be written at the end of each line.

Example:

```
.DATA.L      H'FFFF0000,  
+            H'FF00FF00,  
+            H'FFFFFFFF
```

; In this example, a single source statement is written across three lines.

A comment can be attached at the end of each line.

Example:

```
.DATA.L      H'FFFF0000,   ; Initial value 1.  
+            H'FF00FF00,   ; Initial value 2.  
+            H'FFFFFFFF     ; Initial value 3.
```

; In this example, a comment is attached to each operand.

### 11.1.2 Reserved Words

Reserved words are names that the assembler reserves as symbols with special meanings.

Register names, operators, and the location counter are used as reserved words. Register names are different depending on the target CPU. Refer to the programming manual of the target CPU, for details.

Reserved words must not be used as user symbols.

- Register names  
ER0 to ER7, E0 to E7, R0 to R7, R0H to R7H, R0L to R7L, SP\*, CCR, EXR, MACH, MACL, PC, SBR, VBR
- Operators  
STARTOF, SIZEOF, HIGH, LOW, HWORD, LWORD
- Location counter  
\$

Note: Either ER7 (for the H8SX, H8S/2600, H8S/2000, and H8/300H) or R7 (for the H8/300, and H8/300L) and SP indicate the same register.

### 11.1.3 Symbols

#### (1) Functions of Symbols

Symbols are names defined by the programmer, and perform the following functions.

- Address symbols: Express data storage or branch destination addresses.
- Constant symbols: Express constants.
- Bit data names: Express 1-bit data on memory for bit manipulation instructions.
- Aliases of register names: Express general registers and floating-point registers.
- Section names: Express section names.

The following shows examples of symbol usage.

Examples:

~

```
BRA  SUB1           ;BRA is a branch instruction.
                        ;SUB1 is the address symbol of the destination.
```

~

**SUB1:**

```
-----
MAX:  .EQU  100      ;.EQU is an assembler directive that sets a value to a symbol.
      MOV.B #MAX,R0  ; MAX expresses the constant value 100.
```

~

```
-----
BYSM: .BEQU  1,SYM   ;.BEQU is an assembler directive that sets a value to a bit data.
      BLD    BSYM    ;BSYM indicates bit 1 of SYM
SYM:  .RES.B  1
```

~

```
-----
MIN:  .REG    R0      ;.REG is an assembler directive that defines a register alias.
      MOV.B  #100,MIN ;MIN is an alias for R0.
```

~

```
-----
      .SECTION CD,CODE,ALIGN=2
                        ;.SECTION is an assembler directive that declares a section.
                        ;CD is the name of the current section.
```

~

## (2) Naming Symbols

### (a) Available Characters

The following ASCII characters can be used.

- Alphabetical uppercase and lowercase letters (A to Z, a to z)
- Numbers (0 to 9)
- Underscore (\_)
- Dollar sign (\$)

The assembler distinguishes uppercase letters from lowercase letters in symbols.

### (b) First Character in a Symbol

The first character in a symbol must be one of the following.

- Alphabetical uppercase and lowercase letters (A to Z, a to z)
- Underscore (\_)
- Dollar sign (\$)

Note: The dollar sign character used alone is a reserved word that expresses the location counter.

### (c) Maximum Length of a Symbol

Not limited.

### (d) Names that Cannot Be Used as Symbols

#### (i) Reserved words

Register mnemonic (ER0 to ER7, E0 to E7, R0 to R7, R0H to R7H, R0L to R7L, SP, CCR, EXR, MACH, MACL, PC, SBR, VBR)

Arithmetic operator (STARTOF, SIZEOF, HIGH, LOW, HWORD, LWORD)

Location counter (\$)

#### (ii) Assembler generation symbols

Internal symbol `_$mmmmmm` (*m*: a number from 0 to F)

Structured assembly symbol `_$Innnnn`, `_$Snnnnn`, `_$Fnnnnn`, `_$Wnnnnn`, `_$Rnnnnn`  
(*n*: a number from 0 to 9)

Note: Internal symbols are necessary for assembler internal processing. Internal symbols are not output to assemble listings or object modules.

### (e) Defining and Referencing Symbols

To define a symbol, it must be entered as a label. To reference a symbol, it must be entered as an operand. Symbols that are entered as operands for `.SECTION` or `.MACRO` directives, however, constitute an exception. To reference a symbol (macro name) that has been defined by a `.MACRO` directive, the symbol must be entered as an operation (macro call). A symbol may be referenced before it has been defined. We reference to such as reference as an forward reference. Such references can usually be used, but in some cases they are prohibited.

When a program consists of multiple source files, symbols may be referenced from more than one files. The way a symbol defined in one file is referenced to from another file is called external definition. To reference a symbol that is defined in another file is called external reference. External definitions can be declared by `.EXPORT`, `.GLOBAL`, and `.BEXPORT` directives. External references can be defined by `.IMPORT`, `.GLOBAL`, and `BIMPORT` directives. Be careful with the use of forward and external references, because in some cases, external references such as forward references are prohibited.

## 11.1.4 Constants

### (1) Integer Constants

Integer constants are expressed with a prefix that indicates the radix.

The radix indicator prefix is a notation that indicates the radix of the constant.

- Binary numbers                      The radix indicator “B” plus a binary constant.
- Octal numbers                        The radix indicator “Q” plus an octal constant.
- Decimal numbers                    The radix indicator “D” plus a decimal constant.
- Hexadecimal numbers              The radix indicator “H” plus a hexadecimal constant.

The assembler does not distinguish uppercase letters from lowercase letters in the radix indicator.

The radix indicator and the constant value must be written with no intervening space.

The radix indicator can be omitted. Integer constants with no radix indicator are normally decimal constants, although the radix for such constants can be changed with the .RADIX assembler directive.

Example:

```
.DATA.B B'10001000    ;  
.DATA.B Q'210          ;These source statements express the same  
.DATA.B D'136          ;numerical value.  
.DATA.B H'88           ;
```

Note: "Q" is used instead of "O" to avoid confusion with the digit 0.

### (2) Character Constants

Character constants are considered to be constants that represent ASCII codes.

Character constants are written by enclosing up to four ASCII characters in double quotation marks.

The following ASCII characters can be used in character constants.

ASCII code    {    H'09 (tab)  
                 {    H'20 (space) to H'7E (tilde)

In addition, Japanese characters (shift JIS code or EUC code) and LATIN1 code character can be used. Use two double quotation marks in succession to indicate a single double quotation mark in a character constant. When using Japanese characters in shift JIS code or EUC code, be sure to specify the **sjis** or **euc** command line option, respectively. When using Latin1 code character, be sure to specify the **latin1** command line option. Note that the shift JIS code, EUC code, and LATIN1 code character cannot be used together in one source program.

Example 1:

.DATA.L "**ABC**" ;This is the same as .DATA.L H'00414243.

.DATA.W "**AB**" ;This is the same as .DATA.W H'4142.

.DATA.B "**A**" ;This is the same as .DATA.B H'41.

;The ASCII code for A is: H'41

;The ASCII code for B is: H'42

;The ASCII code for C is: H'43

Example 2:

.DATA.B "''" ;This is a character constant consisting of a single  
;double quotation mark.



### 11.1.5 Location Counter

The location counter expresses the address (location) in memory where the corresponding object code (the result of converting executable instructions and data into code the microprocessor can understand) is stored.

The value of the location counter is automatically adjusted according to the object code output. The value of the location counter can be changed intentionally using assembler directives.

Examples: ~

**.ORG**        H'00001000.        ;This assembler directive sets the location counter to H'00001000

**.DATA.W**    H'FF                ;The object code generated by this assembler directive has  
**.DATA.W**    H'F0                ;a length of 2 bytes.  
                              ;The location counter changes to H'00001002.

**.DATA.W**    H'10                ;The object code generated by this assembler directive has  
                              ;a length of 2 bytes.  
                              ;The location counter changes to H'00001004.  
                              ;The object code generated by this assembler directive has  
                              ;a length of 2 bytes.  
                              ;The location counter changes to H'00001006.  
                              ;ORG is an assembler directive that sets the value of the  
                              location ;counter.  
                              ;.ALIGN is an assembler directive that adjusts the value of  
                              the ;location  
                              ;.DATA is an assembler directive that reserves data in  
                              memory ;counter.  
                              ;.W is a specifier that indicates that data is handled in word  
                              (2 ;bytes) size.  
                              ;.L is a specifier that indicates that data is handled in longword  
                              (4 ;bytes) size.

~

The location counter is referenced using the dollar sign (\$).

Examples:

LABEL1:       .EQU   \$                   ;This assembler directive sets the value of the  
  ;location counter to the symbol LABEL1.  
  ;.EQU is an assembler directive that sets the value to a symbol.

### 11.1.6 Expressions

Expressions are combinations of constants, symbols, and operators that derive a value, and are used as the operands of executable instructions and assembler directives.

#### (1) Elements of Expression

An expression consists of terms, operators, and parentheses.

##### (a) Terms

The terms are the followings:

- A constant
- The location counter (\$)
- A symbol (excluding aliases of the register name)
- The result of a calculation specified by a combination of the above terms and an operator.

An individual term is also a kind of expression.

##### (b) Operators

Table 11.1 shows the operators supported by the assembler.

**Table 11.1 Operators**

Operator Type	Operator	Operation	Coding
Arithmetic operations	+	Unary plus	+ <term>
	–	Unary minus	– <term>
	+	Addition	<term1> + <term2>
	–	Subtraction	<term1> – <term2>
	*	Multiplication	<term1> * <term2>
	/	Division	<term1> / <term2>
Logic operations	~	Unary negation	~ <term>
	&	Logical AND	<term1> & <term2>
		Logical OR	<term1>   <term2>
	~	Exclusive OR	<term1> ~ <term2>
Shift operations	<<	Arithmetic left shift	<term 1> << <term 2>
	>>	Arithmetic right shift	<term 1> >> <term 2>
Section set operations*	STARTOF	Determines the starting address of a section set.	STARTOF <section name>
	SIZEOF	Determines the size of a section set in bytes.	SIZEOF <section name>
Extraction operations	HIGH	Extracts the high-order byte	HIGH <term>
	LOW	Extracts the low-order byte	LOW <term>
	HWORD	Extracts the high-order word	HWORD <term>
	LWORD	Extracts the low-order word	LWORD <term>

Note: HWORD and LWORD cannot be used for the H8/300 or H8/300L.

### (c) Parentheses

Parentheses modify the operation precedence.

#### (d) Operation Precedence

When multiple operations appear in a single expression, the order in which the processing is performed is determined by the operator precedence and by the use of parentheses. The assembler processes operations according to the following rules.

— Rule 1

Processing starts from operations enclosed in parentheses.

When there are nested parentheses, processing starts with the operations surrounded by the innermost parentheses.

— Rule 2

Processing starts with the operator with the highest precedence.

— Rule 3

Processing proceeds in the direction of the operator association rule when operators have the same precedence.

Table 11.2 shows the operator precedence and the association rule.

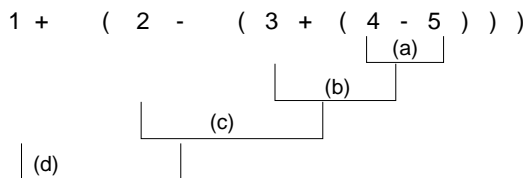
**Table 11.2 Operator Precedence and Association Rules**

Precedence	Operator	Association Rule
1 (high)	+ - ~ STARTOF SIZEOF HIGH LOW HWORD LWORD*	Operators are processed from right to left.
2	* /	Operators are processed from left to right.
3	+ -	Operators are processed from left to right.
4	<< >>	Operators are processed from left to right.
5	&	Operators are processed from left to right.
6 (low)	~	Operators are processed from left to right.

Note: The operators of precedence 1 (highest precedence) are for unary operation.

The figures below show examples of expressions.

Example 1:



The assembler calculates this expression in the order (a) to (d).

The result of (a) is -1

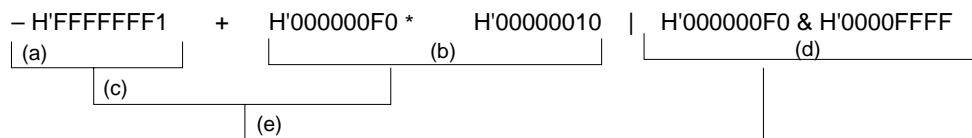
The result of (b) is 2

The result of (c) is 0

The result of (d) is 1

The final result of this calculation is 1.

Example 2:



The assembler calculates this expression in the order (a) to (e).

The result of (a) is H'0000000F

The result of (b) is H'00000F00

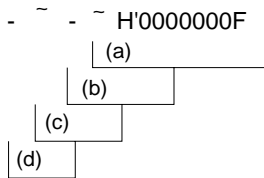
The result of (c) is H'00000F0F

The result of (d) is H'000000F0

The result of (e) is H'00000FFF

The final result of this calculation is H'00000FFF.

### Example 3:



The assembler calculates this expression in the order (a) to (d).

The result of (a) is H'FFFFFFF0

The result of (b) is H'00000010

The result of (c) is H'FFFFFFEF

The result of (d) is H'00000011

} The final result of this calculation is H'00000011.

## (2) Detailed Description on Operation

### (a) STARTOF Operation

Determines the start address of a section set after the specified sections are linked by the optimizing linkage editor.

### (b) SIZEOF Operation

Determines the size of a section set after the specified sections are linked by the optimizing linkage editor.

Example:

```
.CPU      2600A

.SECTION   INIT_RAM,DATA,ALIGN=2

.RES.B H'100

;

.SECTION   INIT_DATA,DATA,ALIGN=2

INIT_BGN.DATA.L      STARTOF INIT_RAM                ; (1)

INIT_END.DATA.L      STARTOF INIT_RAM + SIZEOF INIT_RA ; (2)

;

;

.SECTION   MAIN,CODE,ALIGN=2

INITIAL:

    MOV.L   @INIT_BGN,ER1
    MOV.L   @INIT_END,ER2
    MOV.W   #0,R3
    LOOP:
        CMP.L ER1,ER2
        BEQ   END
        MOV.W R3,@ER1
        ADDS.L #1,ER1
        BRA   LOOP
    END:

    SLEEP

.END
```

Initializes the data area in section  
INIT\_RAM to 0.

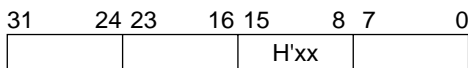
(1) Determines the start address of section INIT\_RAM.

(2) Determines the end address of section INIT\_RAM.

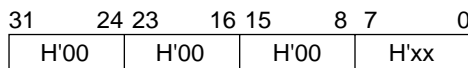
### (c) HIGH Operation

Extracts the high-order byte from the low-order two bytes of a 4-byte value.

Before operation



After operation



Example:

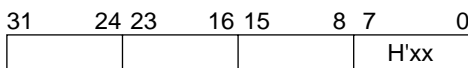
```
LABEL .EQU H'00007FFF
```

```
MOV.W #HIGH LABEL,R0; Assigns H'7F to R0.
```

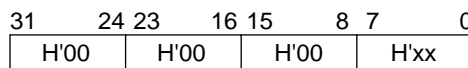
### (d) LOW Operation

Extracts the lowest-order one byte from a 4-byte value.

Before operation



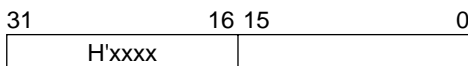
After operation



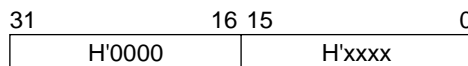
### (e) HWORD Operation

Extracts the high-order two bytes from a 4-byte value.

Before operation



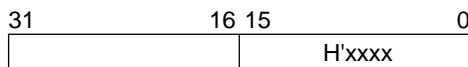
After operation



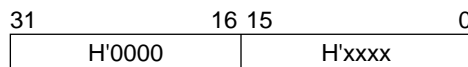
### (f) LWORD Operation

Extracts the low-order two bytes from a 4-byte value.

Before operation



After operation





### (3) Notes on Expressions

#### (a) Internal Processing

The assembler regards expression values as the signed 32-bit signed values regardless of the operand size (8, 16, or 32 bits).

Accordingly, the following example causes an error:

Example:

```
MOV.B  #~H'80:8,R0L
```

The assembler regards H'80 as H'00000080, so the value of ~H'80 is H'FFFFFF7F. Since H'FFFFFF7F is outside the 8-bit value range, it causes an error. To avoid this error, see the following example:

Example:

```
MOV.B  #H'7F:8,R0L      ; The result value of the operation is written directly.
```

```
MOV.B  #~H'80&H'FF:8,R0L ; Low-order bits are validated by using AND
```

```
MOV.B  #LOW ~H'80:8,R0L  ; Lower 8 bits are validated by extracting the low-order bytes
```

### (b) Logic Operators

The logic operators cannot take terms that contain relative values or externally referenced symbols as their operands.

### (c) Arithmetic Operators

Where values must be determined at assembly, the multiplication and division operators cannot take terms that contain relative values or externally referenced symbols as their operands.

Also, a divisor of 0 cannot be used with the division operator.

Example:

```
.IMPORT SYM
```

```
.DATA    SYM/10      ; Correctly assembled.
```

```
.ORG     SYM/10      ; An error will occur.
```

### 11.1.7 String Literal

A string literal is sequences of character data.

The following ASCII characters can be used in strings literal.

ASCII code      $\left\{ \begin{array}{l} \text{H'09 (tab)} \\ \text{H'20 (space) to H'7E (tilde)} \end{array} \right.$

A single character in a string literal has as its value the ASCII code for that character and is represented as a byte sized data object. In addition, Japanese characters in shift JIS code or EUC code, and LATIN1 code character can be used. When using Japanese characters in shift JIS code or EUC code, be sure to specify the `sjis` or `euc` option, respectively. If not specified, Japanese characters are handled as the Japanese code specified by the host computer. When using LATIN1 code character, be sure to specify the **latin1** command line option.

Strings literal must be written enclosed in double quotation marks.

Use two double quotation marks in succession to indicate a single double quotation mark in a string literal.

Examples:

```
.SDATA    "Hello!"           ; This statement reserves the string literal data
                                   ; Hello!

.SDATA    "assembler"       ; This statement reserves the string literal data
                                   ; assembler

.SDATA    " " "Hello!" " "  ; This statement reserves the string literal data
                                   ; " Hello! "
```

;

.SDATA is an assembler directive that reserves string literal data in memory.

Note: The difference between character constants and strings literal is as follows.

Character constants are numeric values. They have a data size of either 1 byte, 2 bytes, or 4 bytes.

Strings literal cannot be handled as numeric values. A string literal has a data size between 1 byte and 255 bytes.

## 11.1.8 Local Label

### (1) Local Label Functions

A local label is valid locally between address symbols. Since a local label does not conflict with the other labels outside its scope, the user does not have to consider other label names. A local label can be defined by writing in the label field in the same way as a normal address symbol, and can be referenced by an operand.

An example of local label descriptions is shown below.

Note: A local label cannot be referenced during debugging.

A local label cannot be specified as any of the following items:

- Macro name
- Section name
- Object module name
- Label in .ASSIGNA, .ASSIGNC, .EQU, .BEQU, .ASSIGN, .REG, or .DEFINE
- Operand in .EXPORT, .IMPORT, .GLOBAL, .BEXPORT, or .BIMPORT

Example:

LABEL1: ; Local block 1 start

```
?0001:    CMP.W    R1,R2
          BEQ      ?0002
          BRA      ?0001
```

?0002:

LABEL2: ; Local block 2 start

```
?0001:    CMP.W    R1,R2
          BGE      ?0002
          BRA      ?0001
```

?0002:

LABEL3:

## (2) Naming Local Labels

### — First Character:

A local label is a string starting with a question mark (?).

### — Usable Characters:

The following ASCII characters can be used in a local label, except for the first character:

- Alphabetical uppercase and lowercase letters (A to Z and a to z)
- Numbers (0 to 9)
- Underscore (\_)
- Dollar sign (\$)

The assembler distinguishes uppercase letters from lowercase ones in local labels.

### — Maximum Length:

The length of local label characters is 2 to 16 characters. If 17 or more characters are specified, the assembler will not recognize them as a local label.

## (3) Scope of Local Labels

The scope of a local label is called a local block. Local blocks are separated by address symbols, or by the `.SECTION` directives.

The local label defined within a local block can be referenced in that local block.

A local label belonging to a local block is interpreted as being unique even if its spelling is the same as local labels in other local blocks; it does not cause an error.

Note: The address symbols defined by the `.ASSIGNA`, `.ASSIGNC`, `.EQU`, `.BEQU`, `.ASSIGN`, or `.REG` directive are not interpreted as delimiters for the local block.

# 11.2 Executable Instructions

## 11.2.1 Overview of Executable Instructions

The executable instructions are the instructions of microprocessor. The microprocessor interprets and executes the executable instructions in the object code stored in memory.

An executable instruction source statement has the following basic form.

[<symbol>:]	Δ<mnemonic>[.<operation size>]	[Δ<addressing mode>[,<addressing mode>]]	[; <comment>]
Label	Operation	Operand	Comment

This section describes the mnemonic, operation size, and addressing mode.

### (1) Mnemonic

The mnemonic expresses the executable instruction. Abbreviations that indicate the type of processing are provided as mnemonics for microprocessor instructions.

The assembler does not distinguish uppercase and lowercase letters in mnemonics.

### (2) Operation Size

The operation size is the unit for processing data. The operation sizes vary with the executable instruction. The assembler does not distinguish uppercase and lowercase letters in the operation size.

Specifier	Data Size
B	Byte (1 byte)
W	Word (2 bytes)
L	Longword (4 bytes)

### (3) Addressing Mode

The addressing mode specifies the data area accessed, and the destination address. The addressing modes vary with the executable instruction.

Table 11.3 lists the addressing modes.

**Table 11.3 Addressing Modes**

<b>Addressing Mode</b>	<b>Name</b>	<b>Description</b>
ERn, Rn, En, RnL, RnH	Register direct	The contents of the specified register.
@ERn, @Rn	Register indirect	A memory location. The value in (E)Rn gives the start address of the memory accessed.
@ERn+, @Rn+, @ERn-, @Rn-	Register indirect with post-increment/decrement	A memory location. The value in ERn (before being incremented*1/decremented*2) gives the start address of the memory accessed. The microprocessor first uses the value in (E)Rn for the memory reference, and increments/decrements (E)Rn afterwards.
@-ERn, @-Rn, @+ERn, @+Rn,	Register indirect with pre-decrement/increment	A memory location. The value in (E)Rn (after being decremented*2/incremented*1) gives the start address of the memory accessed. The microprocessor first decrements/increments (E)Rn, and then uses that value for the memory reference.
@(disp,ERn), @(disp,Rn)	Register indirect with displacement*3	A memory location. The start address of the memory access is given by the value of (E)Rn plus the displacement (disp). The value of (E)Rn is not changed.
@(disp,RnL.B), @(disp,Rn.W), @(disp,ERn.L)	Index register indirect with displacement	A memory location. The start address of the memory access is given by the value of RnL.B/Rn.W/ERn.L plus the displacement (disp). The value of (E)Rn is not changed.
@abs	Absolute address	A memory location. The start address of the memory access is given by the specified absolute address (abs).
#imm	Immediate	Indicates a constant.

Notes: 1. Increment

The amount of the increment is 1 when the operation size is a byte, 2 when the operation size is a word (2 bytes), and 4 when the operation size is a longword (4 bytes).

2. Decrement

The amount of the decrement is 1 when the operation size is a byte, 2 when the operation size is a word, and 4 when the operation size is a longword.

3. Displacement

A displacement is the distance between two points. In this assembly language, the unit of displacement values is in bytes.

**Table 11.3 Addressing Modes (cont)**

<b>Addressing Mode</b>	<b>Name</b>	<b>Description</b>
@@abs	Memory indirect	A memory location. The operand in memory is specified, and its contents are used as the jump address.
@@vec:7	Extended Memory indirect	A memory location. The operand in memory is specified, and its contents are used as the jump address.
@(disp,PC)	PC relative with displacement	A memory location. The start address of the memory access is given by the value of the PC plus the displacement (disp).
@(RnL.B, PC), @(Rn.W, PC), @(ERn.L, PC)	PC relative with index register	A memory location. The start address of the memory access is given by the value of the PC plus RnL.B/Rn.W/ERn.L. The value of (E)Rn is not changed.
<CCR>, <EXR>, <MACH>, <MACL>, <SBR>, <VBR>	Control registers	<CCR>: The internal state of CPU. <EXR>: Trace bit and interrupt mask bits <MACH>, <MACL>: MAC operation results <SBR>: Short address base address <VBR>: Vector base address

### 11.2.2 Notes on Executable Instructions

The operation size that can be specified vary with the mnemonic and the addressing mode combination.

#### (1) H8SX Executable Instruction and Operation Size Combinations:

##### (a) Size of the executable instruction

Table 11.4 shows the H8SX allowable executable instruction and operation size combinations when in the maximum mode, advanced mode, middle mode, or normal mode.



**Table 11.4 H8SX Executable Instruction and Operation Size Combinations**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
ADD	O	O	O	B
ADDS	×	×	O	L
ADDX	O	O	O	B
AND	O	O	O	B
ANDC	O	×	×	B
BAND	O	×	×	B
Bcc	-	-	-	-*1
BCLR	O	×	×	B
BCLR/EQ	O	×	×	B
BCLR/NE	O	×	×	B
BFLD	O	×	×	B
BFST	O	×	×	B
BIAND	O	×	×	B
BILD	O	×	×	B
BIOR	O	×	×	B
BIST	O	×	×	B
BISTZ	O	×	×	B
BIXOR	O	×	×	B
BLD	O	×	×	B
BNOT	O	×	×	B
BOR	O	×	×	B
BRA/BC	-	-	-	-*1
BRA/BS	-	-	-	-*1
BRA/S	-	-	-	-*1
BSET	O	×	×	B
BSET/EQ	O	×	×	B
BSET/NE	O	×	×	B
BSR	-	-	-	-*1
BSR/BC	-	-	-	-*1
BSR/BS	-	-	-	-*1

Note: 1. Size cannot be specified.

**Table 11.4 H8SX Executable Instruction and Operation Size Combinations (cont)**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
BST	O	×	×	B
BSTZ	O	×	×	B
BTST	O	×	×	B
BXOR	O	×	×	B
CLRMAC	-	-	-	-*1, *3
CMP	O	O	O	B
DAA	O	×	×	B
DAS	O	×	×	B
DEC	O	O	O	B
DIVS	×	O	O	W
DIVU	×	O	O	W
DIVXS	O	O	×	B
DIVXU	O	O	×	B
EEPMOV	O	O	×	B
EXTS	×	O	O	W
EXTU	×	O	O	W
INC	O	O	O	B
JMP	-	-	-	-*1
JSR	-	-	-	-*1
LDC	O	O	O	B/L *4
LDM	×	×	O	L
LDMAC	×	×	O	L*3
MAC	-	-	-	-*1, *3

Notes: 1. Size cannot be specified.

3. Valid when specified with a multiplier.

4. If the control register specified is CCR or EXR, B (byte size) or W (word size) can be specified and the default is B.

If the control register specified is SBR or VBR, only L (long word size) can be specified.

**Table 11.4 H8SX Executable Instruction and Operation Size Combinations (cont)**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
MOV	O	O	O	B
MOVA	×	×	O	L*5
MOVFPE	O	×	×	B
MOVMD	O	O	O	B
MOVSD	O	×	×	B
MOVTPE	O	×	×	B
MULS	×	O	O	W
MULS/U	×	×	O	L*3
MULU	×	O	O	W
MULU/U	×	×	O	L*3
MULXS	O	O	×	B
MULXU	O	O	×	B
NEG	O	O	O	B
NOP	-	-	-	-*1
NOT	O	O	O	B
OR	O	O	O	B
ORC	O	×	×	B
POP	×	O	O	*2
PUSH	×	O	O	*2

- Notes: 1. Size cannot be specified.
2. L (longword size) in the maximum mode, advanced mode, or middle mode, and W (word size) in normal mode.
3. Valid when specified with a multiplier.
5. Specify C as an operation size in order to generate an object code of the compact format. With C specified, the assembler will generate the object code using only the destination register number. No error will occur and the register number in the source operand is ignored when the register number in the source operand differs from the destination register number and when C is the operation size.
- MOVA/B.L @(10:16,R1.W),ER1 ; General format. Object code: H'78197A99000A
- MOVA/B.C @(10:16,R1.W),ER1 ; Compact format. Object code: H'7A99000A

**Table 11.4 H8SX Executable Instruction and Operation Size Combinations (cont)**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
ROTL	O	O	O	B
ROTR	O	O	O	B
ROTXL	O	O	O	B
ROTXR	O	O	O	B
RTE	-	-	-	-*1
RTE/L	-	-	-	-*1
RTS	-	-	-	-*1
RTS/L	-	-	-	-*1
SHAL	O	O	O	B
SHAR	O	O	O	B
SHLL	O	O	O	B
SHLR	O	O	O	B
SLEEP	-	-	-	-*1
STC	O	O	O	B/L *4
STM	×	×	O	L
STMAC	×	×	O	L*3
SUB	O	O	O	B
SUBS	×	×	O	L
SUBX	O	O	O	B
TAS	O	×	×	B
TRAPA	-	-	-	-*1
XOR	O	O	O	B
XORC	O	×	×	B

Notes: 1. Size cannot be specified.  
3. Valid when specified with a multiplier.  
4. If the control register specified is CCR or EXR, B (byte size) or W (word size) can be specified and the default is B.  
If the control register specified is SBR or VBR, only L (long word size) can be specified.

(b) Addressing format

The addressing format for the H8SX in maximum mode, advanced mode, or middle mode, and in normal mode is shown in table 11.5.

**Table 11.5 H8SX Series Addressing Format**

Addressing Format	Description <sup>1</sup>
Register direct	{ERn   En   Rn   RnH   RnL}
Register indirect	@ERn
Post-increment register indirect	@ERn+
Post-decrement register indirect	@ERn-
Pre-increment register indirect	@+ERn
Pre-decrement register indirect	@-ERn
Register indirect with displacement	@(disp[ : {2   16   32} ], ERn)
Index register indirect with displacement	@(disp[ : {16   32} ], {RnL.B   Rn.W   ERn.L})
Absolute address	@abs[ : {8   16   24   32} ]
Immediate data	#imm[ : {3   4   5   8   16   32} ]
Memory indirect	@@abs[ : 8]
Extension memory indirect	@@vec:7
Program counter relative with displacement	d[ : {8   16} ]
Program counter index relative	{RnL.B   Rn.W   ERn.L}
Control registers	CCR, EXR, MACH, MACL, SBR, VBR

Notes: 1. n: Register number (0 to 7<sup>2</sup>)

disp: Displacement

abs: Absolute address

imm: Immediate data

vec: Vector address

2. ER7 is the same as SP (stack pointer).

(2) H8S/2600 Executable Instruction and Operation Size Combinations:

(a) Size of the executable instruction

Table 11.6 shows the H8S/2600 allowable executable instruction and operation size combinations when in the advanced mode or normal mode.

**Table 11.6 H8S/2600 Executable Instruction and Operation Size Combinations**

<b>Executable Instructions</b>		<b>Operation Sizes</b>			
<b>Mnemonic</b>	<b>B</b>	<b>W</b>	<b>L</b>	<b>Default when Omitted</b>	
ADD	O	O	O	B	
ADDS	×	×	O	L	
ADDX	O	×	×	B	
AND	O	O	O	B	
ANDC	O	×	×	B	
BAND	O	×	×	B	
Bcc	-	-	-	-*1	
BCLR	O	×	×	B	
BIAND	O	×	×	B	
BILD	O	×	×	B	
BIOR	O	×	×	B	
BIST	O	×	×	B	
BIXOR	O	×	×	B	
BLD	O	×	×	B	
BNOT	O	×	×	B	
BOR	O	×	×	B	
BSET	O	×	×	B	
BSR	-	-	-	-*1	
BST	O	×	×	B	
BTST	O	×	×	B	
BXOR	O	×	×	B	

Note: 1. Size cannot be specified.

**Table 11.6 H8S/2600 Executable Instruction and Operation Size Combinations (cont)**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
CLRMAC	-	-	-	-*1
CMP	O	O	O	B
DAA	O	×	×	B
DAS	O	×	×	B
DEC	O	O	O	B
DIVXS	O	O	×	B
DIVXU	O	O	×	B
EEPMOV	O	O	×	B
EXTS	×	O	O	W
EXTU	×	O	O	W
INC	O	O	O	B
JMP	-	-	-	-*1
JSR	-	-	-	-*1
LDC	O	O	×	B
LDM	×	×	O	L
LDMAC	×	×	O	L
MAC	-	-	-	-*1
MOV	O	O	O	B
MOVFPE	O	×	×	B
MOVTPE	O	×	×	B
MULXS	O	O	×	B
MULXU	O	O	×	B
NEG	O	O	O	B
NOP	-	-	-	-*1
NOT	O	O	O	B
OR	O	O	O	B

Note: 1. Size cannot be specified.

**Table 11.6 H8S/2600 Executable Instruction and Operation Size Combinations (cont)**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
ORC	O	×	×	B
POP	×	O	O	*2
PUSH	×	O	O	*2
ROTL	O	O	O	B
ROTR	O	O	O	B
ROTXL	O	O	O	B
ROTXR	O	O	O	B
RTE	-	-	-	-*1
RTS	-	-	-	-*1
SHAL	O	O	O	B
SHAR	O	O	O	B
SHLL	O	O	O	B
SHLR	O	O	O	B
SLEEP	-	-	-	-*1
STC	O	O	×	B
STM	×	×	O	L
STMAC	×	×	O	L
SUB	O	O	O	B
SUBS	×	×	O	L
SUBX	O	×	×	B
TAS	O	×	×	B
TRAPA	-	-	-	-*1
XOB	O	O	O	B
XOBC	O	×	×	B

Notes: 1. Size cannot be specified.

2. L (longword size) in the advanced mode, and W (word size) in normal mode.



(b) Addressing format

The addressing format for the H8S/2600 in advanced mode and in normal mode is shown in table 11.7.

**Table 11.7 H8S/2600 Series Addressing Format**

Addressing Format	Description <sup>1</sup>
Register direct	{ERn   En   Rn   RnH   RnL}
Register indirect	@ERn
Post-increment register indirect	@ERn+
Pre-decrement register indirect	@-ERn
Register indirect with displacement	@(disp[ : {16   32} ], ERn)
Absolute address	@abs[ : {8   16   24   32} ]
Immediate data	#imm[ : {8   16   32} ]
Memory indirect	@@abs[ : 8]
Program counter relative with displacement	d[ : {8   16} ]
Control registers	CCR, EXR, MACH, MACL

Notes: 1. n: Register number (0 to 7<sup>2</sup>)  
disp: Displacement  
abs: Absolute address  
imm: Immediate data  
2. ER7 is the same as SP (stack pointer).

(3) H8S/2000 Executable Instruction and Operation Size Combinations:

(a) Size of the executable instruction

Table 11.8 shows the H8S/2000 allowable executable instruction and operation size combinations when in the advanced mode or normal mode.

**Table 11.8 H8S/2000 Executable Instruction and Operation Size Combinations**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
ADD	O	O	O	B
ADDS	×	×	O	L
ADDX	O	×	×	B
AND	O	O	O	B
ANDC	O	×	×	B
BAND	O	×	×	B
Bcc	-	-	-	-*1
BCLR	O	×	×	B
BIAND	O	×	×	B
BILD	O	×	×	B
BIOR	O	×	×	B
BIST	O	×	×	B
BIXOR	O	×	×	B
BLD	O	×	×	B
BNOT	O	×	×	B
BOR	O	×	×	B
BSET	O	×	×	B
BSR	-	-	-	-*1
BST	O	×	×	B
BTST	O	×	×	B
BXOR	O	×	×	B

Note: 1. Size cannot be specified.

**Table 11.8 H8S/2000 Executable Instruction and Operation Size Combinations (cont)**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
CMP	O	O	O	B
DAA	O	×	×	B
DAS	O	×	×	B
DEC	O	O	O	B
DIVXS	O	O	×	B
DIVXU	O	O	×	B
EEPMOV	O	O	×	B
EXTS	×	O	O	W
EXTU	×	O	O	W
INC	O	O	O	B
JMP	-	-	-	-*1
JSR	-	-	-	-*1
LDC	O	O	×	B
LDM	×	×	O	L
MOV	O	O	O	B
MOVFPE	O	×	×	B
MOVTPE	O	×	×	B
MULXS	O	O	×	B
MULXU	O	O	×	B
NEG	O	O	O	B
NOP	-	-	-	-*1
NOT	O	O	O	B
OR	O	O	O	B
ORC	O	×	×	B
POP	×	O	O	*2
PUSH	×	O	O	*2

Notes: 1. Size cannot be specified.

2. L (longword size) in the advanced mode, and W (word size) in normal mode.

**Table 11.8 H8S/2000 Executable Instruction and Operation Size Combinations (cont)**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
ROTL	O	O	O	B
ROTR	O	O	O	B
ROTXL	O	O	O	B
ROTXR	O	O	O	B
RTE	-	-	-	-*1
RTS	-	-	-	-*1
SHAL	O	O	O	B
SHAR	O	O	O	B
SHLL	O	O	O	B
SHLR	O	O	O	B
SLEEP	-	-	-	-*1
STC	O	O	×	B
STM	×	×	O	L
SUB	O	O	O	B
SUBS	×	×	O	L
SUBX	O	×	×	B
TAS	O	×	×	B
TRAPA	-	-	-	-*1
XOR	O	O	O	B
XORC	O	×	×	B

Note: 1. Size cannot be specified.

(b) Addressing format

The addressing format for the H8S/2000 in advanced mode and in normal mode is shown in table 11.9.

**Table 11.9 H8S/2000 Series Addressing Format**

Addressing Format	Description <sup>1</sup>
Register direct	{ERn   En   Rn   RnH   RnL}
Register indirect	@ERn
Post-increment register indirect	@ERn+
Pre-decrement register indirect	@-ERn
Register indirect with displacement	@(disp[ : {16   32} ], ERn)
Absolute address	@abs[ : {8   16   24   32} ]
Immediate data	#imm[ : {8   16   32} ]
Memory indirect	@@abs[ : 8]
Program counter relative with displacement	d[ : {8   16} ]
Control registers	CCR, EXR

Notes: 1. n: Register number (0 to 7<sup>2</sup>)  
disp: Displacement  
abs: Absolute address  
imm: Immediate value  
2. ER7 is the same as SP (stack pointer).

(4) H8/300H Executable Instruction and Operation Size Combinations:

(a) Size of the executable instruction

Table 11.10 shows the H8/300H allowable executable instruction and operation size combinations when in the advanced mode or normal mode.

**Table 11.10 H8/300H Executable Instruction and Operation Size Combinations**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
ADD	O	O	O	B
ADDS	×	×	O	L
ADDX	O	×	×	B
AND	O	O	O	B
ANDC	O	×	×	B
BAND	O	×	×	B
Bcc	-	-	-	-*1
BCLR	O	×	×	B
BIAND	O	×	×	B
BILD	O	×	×	B
BIOR	O	×	×	B
BIST	O	×	×	B
BIXOR	O	×	×	B
BLD	O	×	×	B
BNOT	O	×	×	B
BOR	O	×	×	B
BSET	O	×	×	B
BSR	-	-	-	-*1
BST	O	×	×	B
BTST	O	×	×	B
BXOR	O	×	×	B

Note: 1. Size cannot be specified.

**Table 11.10 H8/300H Executable Instruction and Operation Size Combinations (cont)**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
CMP	O	O	O	B
DAA	O	×	×	B
DAS	O	×	×	B
DEC	O	O	O	B
DIVXS	O	O	×	B
DIVXU	O	O	×	B
EEPMOV	O	O	×	B
EXTS	×	O	O	W
EXTU	×	O	O	W
INC	O	O	O	B
JMP	-	-	-	-*1
JSR	-	-	-	-*1
LDC	O	O	×	B
MOV	O	O	O	B
MOVFPE	O	×	×	B
MOVTPE	O	×	×	B
MULXS	O	O	×	B
MULXU	O	O	×	B
NEG	O	O	O	B
NOP	-	-	-	-*1
NOT	O	O	O	B
OR	O	O	O	B
ORC	O	×	×	B
POP	×	O	O	*2
PUSH	×	O	O	*2

Notes: 1. Size cannot be specified.

2. L (longword size) in the advanced mode, and W (word size) in normal mode.

**Table 11.10 H8/300H Executable Instruction and Operation Size Combinations (cont)**

<b>Executable Instructions</b>		<b>Operation Sizes</b>			
<b>Mnemonic</b>	<b>B</b>	<b>W</b>	<b>L</b>	<b>Default when Omitted</b>	
ROTL	O	O	O	B	
ROTR	O	O	O	B	
ROTXL	O	O	O	B	
ROTXR	O	O	O	B	
RTE	-	-	-	-*1	
RTS	-	-	-	-*1	
SHAL	O	O	O	B	
SHAR	O	O	O	B	
SHLL	O	O	O	B	
SHLR	O	O	O	B	
SLEEP	-	-	-	-*1	
STC	O	O	×	B	
SUB	O	O	O	B	
SUBS	×	O	O	L	
SUBX	O	×	×	B	
TRAPA	-	-	-	-*1	
XOR	O	O	O	B	
XORC	O	×	×	B	

Note: 1. Size cannot be specified.



(b) Addressing format

The addressing format for the H8/300H in advanced mode and in normal mode is shown in table 11.11.

**Table 11.11 H8/300H Series Addressing Format**

Addressing Format	Description <sup>1</sup>
Register direct	{ERn   En   Rn   RnH   RnL}
Register indirect	@ERn
Post-increment register indirect	@ERn+
Pre-decrement register indirect	@-ERn
Register indirect with displacement	@(disp[ : {16   24} ], ERn)
Absolute address	@abs[ : {8   16   24} ]
Immediate data	#imm[ : {8   16   32} ]
Memory indirect	@ @abs[ : 8]
Program counter relative with displacement	d[ : {8   16} ]
Control registers	CCR

Notes: 1. n: Register number (0 to 7<sup>2</sup>)  
disp: Displacement  
abs: Absolute address  
imm: Immediate value  
2. ER7 is the same as SP (stack pointer).

(5) H8/300 and H8/300L Executable Instruction and Operation Size Combinations:

(a) Size of the executable instruction

Table 11.12 shows the H8/300 and H8/300L allowable executable instruction and operation size combinations when in the advanced mode or normal mode.

**Table 11.12 H8/300 and H8/300L Executable Instruction and Operation Size Combinations**

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
ADD	O	O	×	B
ADDS	×	O	×	W
ADDX	O	×	×	B
AND	O	×	×	B
ANDC	O	×	×	B
BAND	O	×	×	B
Bcc	-	-	-	-*
BCLR	O	×	×	B
BIAND	O	×	×	B
BILD	O	×	×	B
BIOR	O	×	×	B
BIST	O	×	×	B
BIXOR	O	×	×	B
BLD	O	×	×	B
BNOT	O	×	×	B
BOR	O	×	×	B
BSET	O	×	×	B
BSR	-	-	-	-*
BST	O	×	×	B
BTST	O	×	×	B
BXOR	O	×	×	B

Note: Size cannot be specified.

**Table 11.12 H8/300 and H8/300L Executable Instruction and Operation Size Combinations**  
(cont)

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
CMP	O	O	×	B
DAA	O	×	×	B
DAS	O	×	×	B
DEC	O	×	×	B
DIVXU	O	×	×	B
EEPMOV	-	-	-	-*
INC	O	×	×	B
JMP	-	-	-	-*
JSR	-	-	-	-*
LDC	O	×	×	B
MOV	O	O	×	B
MOVFPE*	O	×	×	B
MOVTPE*	O	×	×	B
MULXU	O	×	×	B
NEG	O	×	×	B
NOP	-	-	-	-*
NOT	O	×	×	B
OR	O	×	×	B
ORC	O	×	×	B
POP	×	O	×	W
PUSH	×	O	×	W

Note: Size cannot be specified.

**Table 11.12 H8/300 and H8/300L Executable Instruction and Operation Size Combinations**  
(cont)

Executable Instructions	Operation Sizes			
	B	W	L	Default when Omitted
ROTL	O	×	×	B
ROTR	O	×	×	B
ROTXL	O	×	×	B
ROTXR	O	×	×	B
RTE	-	-	-	-*
RTS	-	-	-	-*
SHAL	O	×	×	B
SHAR	O	×	×	B
SHLL	O	×	×	B
SHLR	O	×	×	B
SLEEP	-	-	-	-*
STC	O	×	×	B
SUB	O	O	×	B
SUBS	×	O	×	W
SUBX	O	×	×	B
XOR	O	×	×	B
XORC	O	×	×	B

Note: Size cannot be specified.

(b) Addressing format

The addressing format for the H8/300 and H8/300L in advanced mode and in normal mode is shown in table 11.13.

**Table 11.13 H8/300 and H8/300L Series Addressing Format**

Addressing Format	Description <sup>1</sup>
Register direct	{Rn   RnH   RnL}
Register indirect	@Rn
Post-increment register indirect	@Rn+
Pre-decrement register indirect	@-Rn
Register indirect with displacement	@(disp[ : 16], Rn)
Absolute address	@abs[ : {8   16} ]
Immediate data	#imm[ : {8   16} ]
Memory indirect	@ @abs[ : 8]
Program counter relative with displacement	d[ : 8]
Control registers	CCR

Notes: 1. n: Register number (0 to 7<sup>2</sup>)

disp: Displacement

abs: Absolute address

imm: Immediate value

2. R7 is the same as SP (stack pointer).

## 11.3 Assembler Directives

The assembler directives are instructions that the assembler interprets and executes. . The underscores indicate the default. Table 11.14 lists the assembler directives provided by this assembler.

**Table 11.14 Assembler Directives**

Type	Mnemonic	Function
Target CPU	.CPU	Specifies the target CPU.
8-bit short absolute area	.SBR	Specifies the origin of the 8-bit short absolute address area.
Section and the location counter	.SECTION	Declares a section.
	.ORG	Sets the value of the location counter.
	.ALIGN	Corrects the value of the location counter to a multiple of boundary alignment value.
Symbols	.EQU	Sets a symbol value.
	.ASSIGN	Sets or resets a symbol value.
	.REG	Defines the alias of a register name.
	.BEQU	Defines a bit data name.
Data and data area reservation	.DATA	Reserves integer data.
	.DATAB	Reserves an integer data block.
	.SDATA	Reserves string literal data.
	.SDATAB	Reserves a string literal data block.
	.SDATAC	Reserves string literal data (with length).
	.SDATAZ	Reserves string literal data (with zero terminator).
	.RES	Reserves data area.
	.SRES	Reserves string literal data area.
	.SRESC	Reserves string literal data area (with length).
	.SRESZ	Reserves string literal data area (with zero terminator).

**Table 11.14 Assembler Directives (cont)**

Type	Mnemonic	Function
Externally defined and externally referenced symbol	.EXPORT	Declares externally defined symbols.
	.IMPORT	Declares externally referenced symbols.
	.GLOBAL	Declares externally defined and externally referenced symbols.
	.BEXPORT	Declares externally defined symbol BEQU.
	.BIMPORT	Declares externally referenced symbol BEQU.
	.ABS8	Specifies the 8-bit short absolute address symbol.
	.NOABS8	Disables specifying the 8-bit short absolute address symbol.
Object modules	.OUTPUT	Controls object module and debugging information output.
	.DEBUG	Controls the output of symbolic debugging information.
	.LINE	Changes the file name and line number for the debugging information.
	.DISPSIZE	Sets the displacement size.
Assemble listing	.PRINT	Controls assemble listing output.
	.LIST	Controls the output of the source program listing.
	.FORM	Sets the number of lines and columns in the assemble listing.
	.HEADING	Sets the header for the source program listing.
	.PAGE	Inserts a new page in the source program listing.
	.SPACE	Outputs blank lines to the source program listing.
Other directives	.PROGRAM	Sets the name of the object module.
	.RADIX	Sets the radix in which integer constants with no radix specifier are interpreted.
	.END	Specifies an entry point and the end of the source program.
	.STACK	Defines the stack value for the specified symbol.

## .CPU

Description Format: Δ.CPUΔ<target CPU>

```
<target CPU>={ AE5 |  
                H8SXX [ :<bit width of the address space>] [ :{M|D|MD}} |  
                H8SXA [ :<bit width of the address space>] [ :{M|D|MD}} |  
                H8SXM [ :<bit width of the address space>] [ :{M|D|MD}} |  
                H8SXN [ :{M|D|MD}} |  
                2600A [ :<bit width of the address space>] |  
                2600N |  
                2000A [ :<bit width of the address space>] |  
                2000N |  
                300HA [ :<bit width of the address space>] |  
                300HN |  
                300 | 300L }
```

The label field is not used.

Description: .CPU specifies the CPU type and the operating mode for the object program to be generated, the bit width of the address area, and whether or not the multiplier and/or divider exist.

The bit width of the address area can be specified only in the maximum mode, advanced mode, and middle mode.

The target CPU and the bit width of the address area are as follows:



Suboption Name	Description
AE5	Creates an object for the AE5.
H8SXX[:<bit width of the address space>] [:{M D MD}]	Creates an object for the H8SX maximum mode. <bit width of the address space> is 28 or 32, which is 256 Mbytes or 4 Gbytes, respectively. <bit width of the address space> is 32 by default. A multiplier and/or a divider can be specified.
H8SXA [:<bit width of the address space>] [:{M D MD}]	Creates an object for the H8SX advanced mode. <bit width of the address space> is 20, 24, 28, or 32, which is 1 Mbyte, 16 Mbytes, 256 Mbytes, or 4 Gbytes, respectively. <bit width of the address space> is 24 by default. A multiplier and/or a divider can be specified.
H8SXM [:<bit width of the address space>] [:{M D MD}]	Creates an object for the H8SX middle mode. <bit width of the address space> is 20 or 24, which is 1 Mbyte or 16 Mbytes, respectively. <bit width of the address space> is 24 by default. A multiplier and/or a divider can be specified.
H8SXN [:{M D MD}]	Creates an object for the H8SX normal mode. A multiplier and/or a divider can be specified.
2600A[:<bit width of the address space>]	Creates an object for the H8S/2600 advanced mode. <bit width of the address space> is 20, 24, 28, or 32, which is 1 Mbyte, 16 Mbytes, 256 Mbytes, or 4 Gbytes, respectively. <bit width of the address space> is 24 by default.
2600N	Creates an object for the H8S/2600 normal mode.
2000A[:<bit width of the address space>]	Creates an object for the H8S/2000 advanced mode. <bit width of the address space> is 20, 24, 28, or 32, which is 1 Mbyte, 16 Mbytes, 256 Mbytes, or 4 Gbytes, respectively. <bit width of the address space> is 24 by default.
2000N	Creates an object for the H8S/2000 normal mode.
300HA[:<bit width of the address space>]	Creates an object for the H8/300H advanced mode. <bit width of the address space> is 20 or 24, which is 1 Mbyte or 16 Mbytes, respectively. <bit width of the address space> is 24 by default.
300HN	Creates an object for the H8/300H normal mode.
300	Creates an object for the H8/300.
300L	Creates an object for the H8/300L.

Specify whether or not a multiplier and/or a divider exist as follows:

Multiplier/Divider	Specification Method
Without multiplier and without divider	No specification
With multiplier and without divider	M
Without multiplier and with divider	D
With multiplier and with divider	MD

Use MAC, LDMAC, STMAC, CLRMAC, MULU/U, or MULS/U as an additional instruction with a multiplier.

There are no additional instructions with a divider.

Specify this directive at the beginning of the source program. If it is not specified at the beginning, an error will occur. However, directives related to assembly listing can be written before this directive.

When several .CPU directives are specified, only the first specification becomes valid.

The assembler gives priority to target CPU specification in the order of cpu option, .CPU directive, and the H38CPU environment variable.

If the directive is not specified, the CPU selected by the environment variable H38CPU becomes valid.

Example:

<b>.CPU</b>	2600A:20	;Assembles program for 1 Mbyte
<b>SECTION</b>	A,CODE,ALIGN=2	; of H8S/2600 advanced mode.
<b>MOV.L</b>	ER0,ER1	
<b>MOV.L</b>	ER0,ER2	

## **.SBR**

Description Format:  $\Delta$ .SBR $\Delta$ [<constant>]

The label field is not used.

Description: .SBR declares the origin of the 8-bit short absolute address area.  
When .SBR <constant> is specified, the specified constant value is the origin of the 8-bit short absolute address area. The lower 8 bits of the origin must be 0.

When only .SBR is specified without <constant>, the origin of the 8-bit short absolute address area differs depending on whether or not the SBR option is specified. When the SBR option is specified, the origin is specified with the SBR option. When the SBR option is not specified, the origin is as shown below depending on the bit width of the address space.

CPU/Operating Mode		Origin of the 8-Bit Short Absolute Address
H8SX maximum mode	H8SXX[:32]	H'FFFFFF00
	H8SXX:28	H'0FFFFFF00
H8SX advanced mode	H8SXA:32	H'FFFFFF00
	H8SXA:28	H'0FFFFFF00
	H8SXA[:24]	H'00FFFF00
	H8SXA:20	H'000FFF00
H8SX middle mode	H8SXM[:24]	H'00FFFF00
	H8SXM:20	H'000FFF00
H8SX normal mode	H8SXN	H'0000FF00

When the CPU is H8SXN, H8SXM, H8SXA, or H8SXX, the SBR directive can be specified.

To set an address to SBR (short address base register), the LDC instruction must be described.

Example:

```
.CPU      H8SXA:32
.SECTION  A, CODE, ALIGN=2
.SBR      H'10000          ;Declares H'00010000 as SBR.
MOV.L     #H'10000, ER1
LDC.L     ER1, SBR         ;Sets an address to SBR.
~
MOV.B     @H'FFFFFF00, R0L ;Selects @aa:16.
```

MOV.B	@H'00010050,R0H	;Selects @aa:8.
~		
.SBR		;Clears a declaration of SBR.
MOV.L	#H'FFFFFF00,ER1	
LDC.L	ER1,SBR	;Returns SBR to default value.
~		
MOV.B	@H'FFFFFF00,R0L	;Selects @aa:8.
MOV.B	@H'00010050,R0H	;Selects @aa:32.
~		

## **.SECTION**

Description Format:  $\Delta$ .SECTION $\Delta$ <section name> [,<section attribute> [,<section type>]]

<section attribute>={ CODE | DATA | STACK | DUMMY }  
<section type>={LOCATE= <start address>|ALIGN=<boundary alignment value>}

The label field is not used.

Description: .SECTION is the section declaration assembler directive.  
A section is a part of a program, and the linkage editor regards it as a unit of processing.

### (1) Start of a section

The rules for section names are the same as the rules for symbols. The assembler distinguishes uppercase and lowercase letters.

Attribute	Section Type
CODE	Code section
DATA	Data section
STACK	Stack section
DUMMY	Dummy section

Use locate=<start address> to output an object in an absolute address format.  
Use align=<boundary alignment value> to output an object in a relative address format. The linkage editor will adjust the start address of the section to be the multiple of the boundary alignment value. When the format type is not specified, align=2 is assumed.

Absolute Address Format: The start address of a section is set. The maximum start address is shown below.

<b>CPU/Operating Mode</b>		<b>Maximum Value</b>
H8SX maximum mode	H8SXX[:32]	H'FFFFFFFF
	H8SXX:28	H'0FFFFFFF
H8SX advanced mode	H8SXA:32	H'FFFFFFFF
	H8SXA:28	H'0FFFFFFF
	H8SXA[:24]	H'00FFFFFF
	H8SXA:20	H'000FFFFF
H8SX middle mode	H8SXM[:24]	H'00FFFFFF
	H8SXM:20	H'000FFFFF
H8SX normal mode	H8SXN	H'0000FFFF
H8S/2600 advanced mode	2600A:32	H'FFFFFFFF
	2600A:28	H'0FFFFFFF
	2600A[:24]	H'00FFFFFF
	2600A:20	H'000FFFFF
H8S/2600 normal mode	2600N	H'0000FFFF
H8S/2000 advanced mode	2000A:32	H'FFFFFFFF
	2000A:28	H'0FFFFFFF
	2000A[:24]	H'00FFFFFF
	2600A:20	H'000FFFFF
H8S/2000 normal mode	2000N	H'0000FFFF
H8/300H advanced mode	300HA[:24]	H'00FFFFFF
	300HA:20	H'000FFFFF
H8/300H normal mode	300HN	H'0000FFFF
H8/300	300	H'0000FFFF
H8/300L	300L	H'0000FFFF

Relative Address Format: Boundary alignment value is set.

The linkage editor will adjust the start address of the section to be the multiple of the boundary alignment value.

The values allowed for the boundary alignment value are powers of 2

The assembler provides a default section for the following cases:

- The use of executable instructions when no section has been declared.
- The use of data reservation assembler directives when no section has been declared.
- The use of the .ALIGN directive when no section has been declared.
- The use of the .ORG directive when no section has been declared.
- Reference to the location counter when no section has been declared.

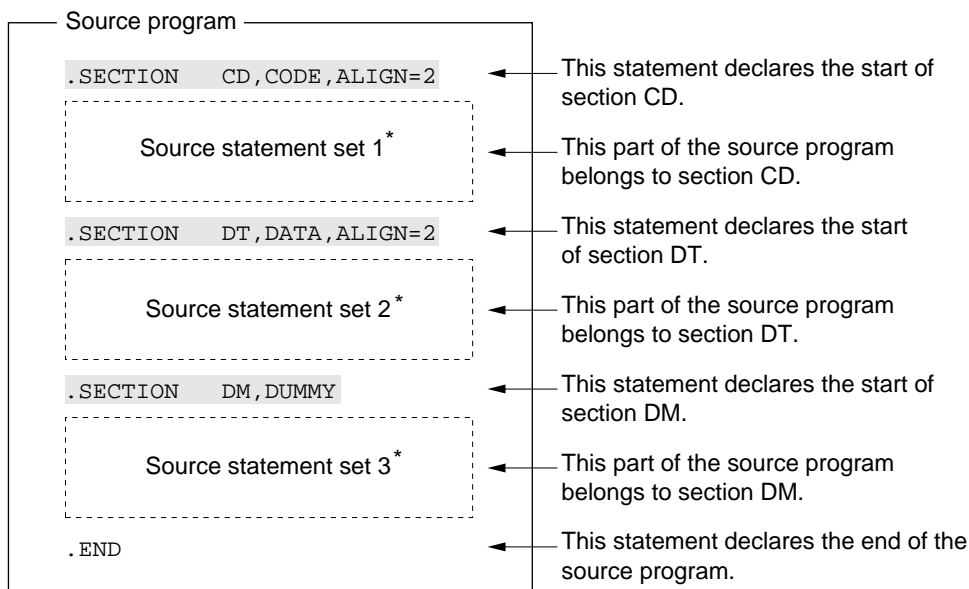
- The use of statements consisting of only the label field when no section has been declared.

## (2) Restart of the section

It is possible to redeclare (and thus restart,) a section that was previously declared in the same file.

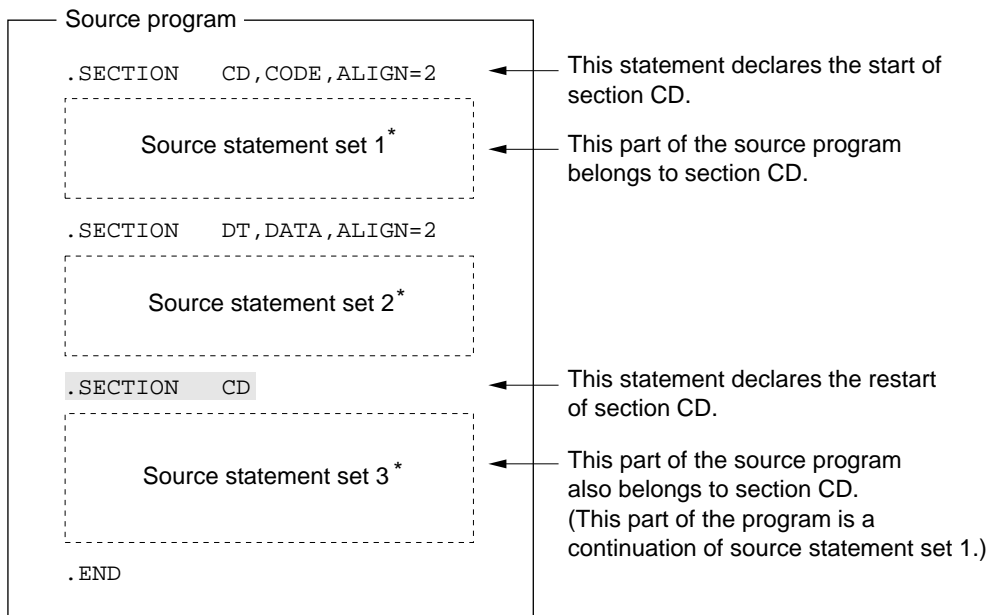
To restart a section, specify a section name that already exists.

The following is a simple example of section declaration.



**Note:** This example assumes that the `.SECTION` directive does not appear in any of the source statement sets 1 to 3.

The following is a simple example of section restart.



Note: This example assumes that the `.SECTION` directive does not appear in any of the source statement sets 1 to 3.



## Example:

```
.ALIGN      2
.DATA.W     H' 0102,H' 0304
~
```

; This section of the program belongs to the default section P.  
; The default section P is a code section, and is a relative  
; address section with a boundary alignment value of 2.

```
.SECTION CD, CODE, ALIGN=2
```

```
MOV        R0,R1
MOV        R0,R2
~
```

; This section of the program belongs to the section CD.  
; The section CD is a code section, and is a relative address  
; section with a boundary alignment value of 2.

```
.SECTION DT, DATA, LOCATE=H'00001000
```

```
X1:        .DATA.W     H' 2222
           .DATA.W     H' 3333
~
```

; This section of the program belongs to the section DT.  
; The section DT is a data section, and is an absolute address  
; section with a start address of H'00001000.

```
.END
```

Note: This example assumes the .SECTION directive does not appear in the parts indicated by "~".

## **.ORG**

Description Format:  $\Delta$ .ORG $\Delta$ <location-counter value>

The label field is not used.

Description: .ORG sets the value of the location counter. The .ORG directive is used to place executable instructions or data at a specific address. The location-counter value must be specified as follows:

- The specification must be a constant value or an address within the section, and,
- Forward reference symbols must not appear in the specification.

The maximum start address is shown below.

CPU/Operating Mode		Maximum Value
H8SX maximum mode	H8SXX[:32]	H'FFFFFFFF
	H8SXX:28	H'0FFFFFFFF
H8SX advanced mode	H8SXA:32	H'FFFFFFFF
	H8SXA:28	H'0FFFFFFFF
	H8SXA[:24]	H'00FFFFFFFF
	H8SXA:20	H'000FFFFFF
H8SX middle mode	H8SXM[:24]	H'00FFFFFFFF
	H8SXM:20	H'000FFFFFF
H8SX normal mode	H8SXN	H'0000FFFF
H8S/2600 advanced mode	2600A:32	H'FFFFFFFF
	2600A:28	H'0FFFFFFFF
	2600A[:24]	H'00FFFFFFFF
	2600A:20	H'000FFFFFF
H8S/2600 normal mode	2600N	H'0000FFFF
H8S/2000 advanced mode	2000A:32	H'FFFFFFFF
	2000A:28	H'0FFFFFFFF
	2000A[:24]	H'00FFFFFFFF
	2600A:20	H'000FFFFFF
H8S/2000 normal mode	2000N	H'0000FFFF
H8/300H advanced mode	300HA[:24]	H'00FFFFFFFF
	300HA:20	H'000FFFFFF
H8/300H normal mode	300HN	H'0000FFFF
H8/300	300	H'0000FFFF
H8/300L	300L	H'0000FFFF

When the location-counter value is specified with an absolute address format, the following condition must be satisfied:

<location-counter value> ≥ <section start address>

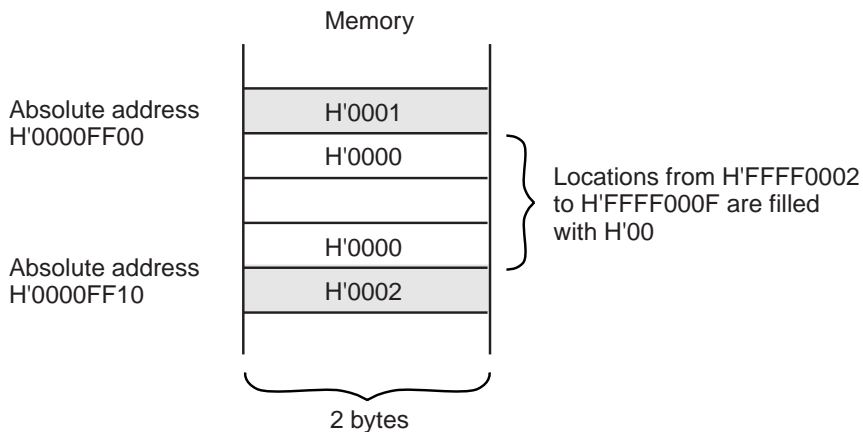
The assembler handles the value of the location counter as follows:

- An absolute address value within an absolute address section.
- A relative address value (relative distance from the section head) within a relative address section.

Example:     `.SECTION DT,DATA,LOCATE=H'0000FF00`  
               `.DATA.W     H'0001`  
               `.ORG     H'0000FF10`     ; This statement sets the value of the location  
   ; counter.  
               `.DATA.L     H'0002`     ; The integer data H'0002 is stored at  
   ; absolute address H'0000FF10.

~

Explanatory Figure for the Coding Example



## **.ALIGN**

Description Format:  $\Delta$ .ALIGN $\Delta$ <boundary alignment value>

The label field is not used.

Description: .ALIGN corrects the location-counter value to be a multiple of the boundary alignment value. Executable instructions and data can be allocated on specific boundary values (address multiples) by using the .ALIGN directive. The location counter value must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

The values allowed for the boundary alignment value are powers of 2. The boundary alignment value differs depending on the specification of the address area.

When .ALIGN is used in a relative section the following must be satisfied:

Boundary alignment value specified by .SECTION  $\geq$  Boundary alignment value specified by .ALIGN

When .ALIGN is used in a code section, the assembler inserts NOP instructions in the object code\* to adjust the value of the location counter. Odd byte size areas are filled with H'00.

When .ALIGN is used in a data, dummy, or stack section, the assembler only adjusts the value of the location counter and does not insert an object code on the memory.

Note: This object code is not displayed in the assemble listing.

```

Example:  .CPU      2600A
          .SECTION  P,DATA,ALIGN=2 ; [1]
          .DATA.B   H'11           ; [2]
          .ALIGN    2              ; [3]
          .DATA.W   H'2222

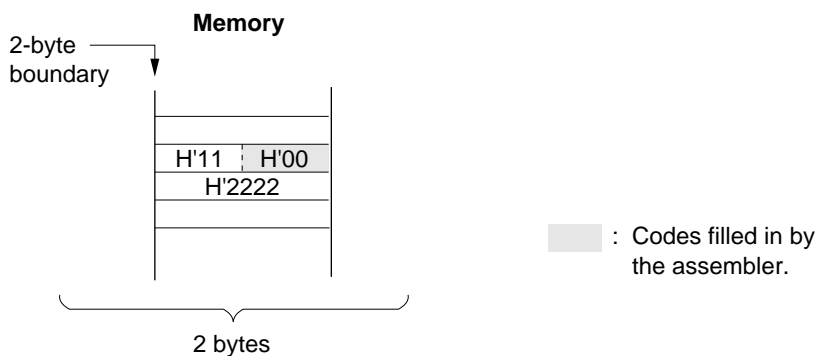
```

~

- [1]: This statement adjusts the start address of the relative address section to be a multiple of 2 at object module linkage.
- [2]: The location counter value of the next data becomes the odd address to secure the one-byte data.
- [3]: This statement adjusts the location counter value to be a multiple of 2 (even address).

### Explanatory Figure for the Coding Example

This example assumes that the byte-sized integer data H'11 is originally located at the 2-byte boundary address. The assembler will insert the filler data as shown in the figure below.



## **.EQU**

Description Format: <symbol>[:] $\Delta$ .EQU $\Delta$ <symbol value>

Description: .EQU sets a value to a symbol.  
Symbols defined with the .EQU directive cannot be redefined.  
The symbol value must be specified as follows:

- The specification must be a constant value, an address value, or an externally referenced symbol value\* and,
- Forward reference symbols must not appear in the specification.

The values allowed for the symbol value are from H'00000000 to H'FFFFFFF.

Note: An externally referenced symbol, externally referenced symbol + constant, or externally referenced symbol – constant can be specified.

Example:

```
~  
  
X1: .EQU      10      ;The value 10 is set to X1.  
X2: .EQU      20      ;The value 20 is set to X2.  
  
    CMP.W     #X1,R0   ;This is the same as CMP.W #10,R0.  
    BNE       LABEL1  
    CMP.W     #X2,R0   ;This is the same as CMP.W #20,R0.  
    BEQ       LABEL2  
  
~
```

## **.ASSIGN**

Description Format: <symbol>[:] $\Delta$ .ASSIGN $\Delta$ <symbol value>

Description: .ASSIGN sets a value to a symbol.  
Symbols defined with the .ASSIGN directive can be redefined with the .ASSIGN directive.  
The symbol value must be specified as follows:

- The specification must be a constant value or an address value, and,
- Forward reference symbols must not appear in the specification.

The values allowed for the symbol value are from H'00000000 to H'FFFFFFF.  
Definitions with the .ASSIGN directive are valid from the point of the definition the program.  
Symbols defined with .ASSIGN have the following limitations:

- They cannot be used as externally defined or externally referenced symbols.
- They cannot be referenced from the debugger.

Example:

```
~  
X1: .ASSIGN 1  
X2: .ASSIGN 2  
    CMP.W  #X1,R0    ;This is the same as CMP.W #1,R0.  
    BNE    LABEL1  
    CMP.W  #X2,R0    ;This is the same as CMP.W #2,R0.  
    BEQ    LABEL2  
  
~
```

```
X1: .ASSIGN 3  
X2: .ASSIGN 4  
    CMP.W  #X1,R0    ; This is the same as CMP.W #3,R0.  
    BNE    LABEL3  
    CMP.W  #X2,R0    ; This is the same as CMP.W #4,R0.  
    BEQ    LABEL4  
  
~
```



## **.REG**

Description Format: <symbol>[:] $\Delta$ .REG $\Delta$ (<register name>)

Description: .REG defines the alias of a register name.

.REG can be specified in two ways:

- Single register:

An alias is defined for one register. It can be specified in any place that the register can be used. A general register can be specified.

- Multiple register:

An alias is defined for two or more registers. This is only available for the CPU type of H8SX series, H8S/2600 series, or H8S/2000 series. It can be specified for the operand of the LDM instruction, STM instruction, or .REG directive. In the H8SX series, it can also be specified for the operand of the RTS/L or RTE/L instruction. A 32-bit general register can be specified for the register name.

Register specification is as follows:

Specification	Description	Example
Single register	Specify either R0L to R7L, R0H to R7H, R0 to R7, E0 to E7, or ER0 to ER7.	SINGLEREG .REG (R0) An alias SINGLEREG is defined for register R0.
Multiple register	Specify more than one register at once by delimiting them with hyphen (-). If the left register number is smaller than the right register number, an error occurs and .REG directive is ignored.	RNG1 REG (ER0-ER3) An alias RNG1 is defined for four registers ER0, ER1, ER2, and ER3.  RNG2 .REG (ER3-ER0) An error occurs because ER0 on the right is smaller than ER3 on the left.*4
Redefining alias of register	Specify an already defined register alias for an operand.	ER00 .REG (ER0-ER3) ER01 .REG (ER00) An alias ER01 is defined for four registers ER0 to ER3.

- Notes: 1. The alias of a register name defined with .REG cannot be redefined.
2. Definitions with the .REG directive are valid from the point of the definition forward in the program.
3. Symbols defined with .REG have the following limitations:  
They cannot be used as externally defined or externally referenced symbols.  
They cannot be referenced from the debugger.
4. The combination of registers specified in the H8SX series is as follows: (ERn-ERn+1; n = 0 to 6), (ERn-ERn+2; n = 0 to 5), (ERn-ERn+3; n = 0 to 4).  
The combination of registers specified in the H8S/2600 series and H8S/2000 series is as follows: (ER0-ER1), (ER2-ER3), (ER4-ER5), (ER6-ER7), (ER0-ER2), (ER4-ER6), (ER0-ER3), (ER4-ER7).

Example:

```
.CPU      2600A
RLST1:    .REG      (R0)
RLST2:    .REG      (ER0-ER2)
          MOV.W      RLST1,@ER6      ; [1]
          LDM.L      @SP+,(RLST2)    ; [2]
          STM.L      (RLST2),@-SP
```

[1]: Defines register R0 as RLST1

[2]: Defines RLST2 to three registers ER0, ER1, and ER2.

## **.BEQU**

Description Format: <symbol>[:]**Δ**.BEQU**Δ**<bit number>, <replaced symbol name>

Description: **.BEQU** specifies a name for one-bit data which is on the memory where bit manipulation is enabled.

The bit data name can be specified at the operand of the bit manipulation instruction.

The specified bit name is replaced by the #xx,@aa format.

The bit number is specified as follows:

- The specification must be a constant value or an address value, and,
- Forward reference symbols must not appear in the specification.

A value from 0 to 7 can be specified for a bit number.

Specify the replaced symbol as follows:

<b>CPU Type</b>	<b>Replaced Symbol Name</b>
H8SX series	8-bit absolute address format (@aa:8)
H8S/2600 series	16-bit absolute address format (@aa:16)
H8S/2000 series	32-bit absolute address format (@aa:32)
H8/300H series	8-bit absolute address format (@aa:8)
H8/300 series	
H8/300L series	

Notes: 1. Specifications with the **.BEQU** directive are valid from the point of specification forward in the program.

2. A symbol defined by the **.BEQU** directive can be externally defined or externally referenced by the **.BEXPORT** and **.BIMPORT** symbols.

Example:

```
.CPU      2600A:32
AD1       .EQU      H'FFFFFF00
AD2       .EQU      H'FFFF8000
AD1B0     .BEQU     0,AD1
AD1B1     .BEQU     1,AD1
AD2B2     .BEQU     2,AD2
AD2B3     .BEQU     3,AD2
```

```
.SECTION   A,CODE,ALIGN=2
BSET.B    AD1B0      ; BSET,B    #0,@AD1:8
BSET.B    AD1B1      ; BSET,B    #1,@AD1:8
BSET.B    AD2B2      ; BSET,B    #2,@AD2:16
BSET.B    AD2B3      ; BSET,B    #3,@AD2:16
```

Bit data name are as follows:

AD1B0: Bit 0 at address H'FFFFFF00

AD1B1: Bit 1 at address H'FFFFFF00

AD2B2: Bit 2 at address H'FFFF8000

AD2B3: Bit 3 at address H'FFFF8000

Supplement:

Bit manipulation instructions that can specify bit data are as follows:  
BSET, BCLR, BNOT, BTST, BAND, BIAND, BOR, BIOR, BXOR,  
BIXOR, BLD, BILD, BST, and BIST

## **.DATA**

Description Format: [<symbol>[:]]Δ.DATA[.<operation size>]Δ<integer data>[,...]

<operation size>: { B | W | L }

Description: .DATA reserves integer data in memory.  
The operation size and the range of integer data are as follows:

Operation Size	Data Size	Integer Data Range
<u>B</u> (byte)	1 byte	H'00000000 to H'000000FF (0 to 255) H'FFFFFF80 to H'FFFFFFF (−128 to −1)
W (word)	2 bytes	H'00000000 to H'0000FFFF (0 to 65,535) H'FFFF8000 to H'FFFFFFF (−32,768 to −1)
L (longword)	4 bytes	H'00000000 to H'FFFFFFF (0 to 4,294,967,295) H'80000000 to H'FFFFFFF (−2,147,483,648 to −1)

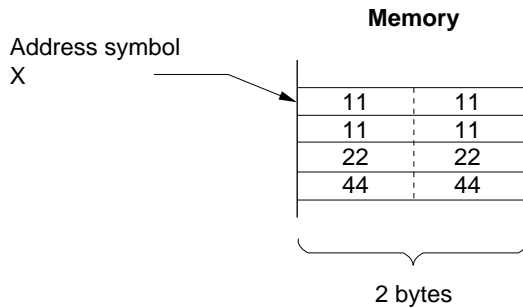
Note: Numbers in parentheses are decimal.

The .DATA.B (byte size) is used when the operation size is omitted.  
Arbitrary values, including relative values, forward referenced symbols and externally referenced symbols, can be used to specify the integer data.  
The operation size determines the range of the integer data that can be specified.

Example:

```
.SECTION A,DATA,ALIGN=2
X:  .DATA.L  H'11111111 ;
    .DATA.W  H'2222      ; These statements reserve integer data.
    .DATA.B  H'44,H'55   ;
```

Explanatory Figure for the Coding Example



Note: The data in this figure is hexadecimal.

## .DATAB

Description Format: [<symbol>[:]]Δ.DATAB[.<operation size>]Δ<block count>,<integer data>

<operation size>: { B | W | L }

Description: .DATAB reserves the specified number of integer data for block count in memory.

The operation size determines the size of the reserved data.

The DATAB.B (byte size) is used when the operation size is omitted.

The block count must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Operation Size	Data Size	Block Size Range*
<u>B</u> (byte)	1 byte	H'00000001 to H'FFFFFFF (1 to 4,294,967,295)
W (word)	2 bytes	H'00000001 to H'7FFFFFF (1 to 2,147,483,647)
L (longword)	4 bytes	H'00000001 to H'3FFFFFF (1 to 1,073,741,823)

Note: Numbers in parentheses are decimal.

Arbitrary values, including relative values, forward reference symbols, and externally referenced symbols, can be used to specify the integer data.

The operation size and the range of block size are as follows:

Operation Size	Integer Data Range*
<u>B</u>	H'00000000 to H'000000FF (0 to 255) H'FFFFFF80 to H'FFFFFFF (-128 to -1)
W	H'00000000 to H'0000FFFF (0 to 65,535) H'FFF8000 to H'FFFFFFF (-32,768 to -1)
L	H'00000000 to H'FFFFFFF (0 to 4,294,967,295) H'80000000 to H'FFFFFFF (-2,147,483,648 to -1)

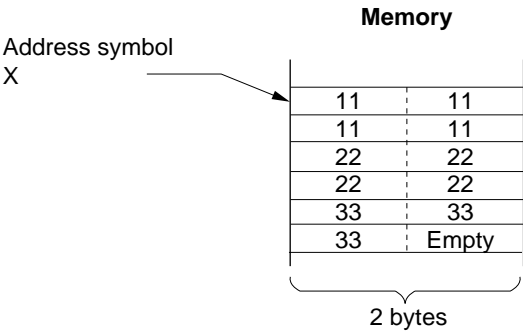
Note: Numbers in parentheses are decimal.

Example:

```
.SECTION      A,DATA,ALIGN=2
X:  .DATAB.L  1,H'11111111      ;
    .DATAB.W  2,H'2222          ; This statement reserves two blocks
    .DATAB.B  3,H'33            ; of integer data.
```

~

Explanatory Figure for the Coding Example



Note: The data in this figure is hexadecimal.



**.SDATA**

Description Format: [`<symbol>[:] $\Delta$ .SDATA $\Delta$ "<string literal>"`[,...]

Description: .SDATA reserves string literal data in memory.  
When specifying a string literal, enclose the character with double quotation marks (“”). When a double quotation mark is used as a character, specify two double quotation marks.  
A control character can be appended to a string literal. Enclose the string literal with double quotation marks and then enclose the control code with angle brackets (<>).

"<string literal>"< control code>

The control code for a control character must be specified as follows:

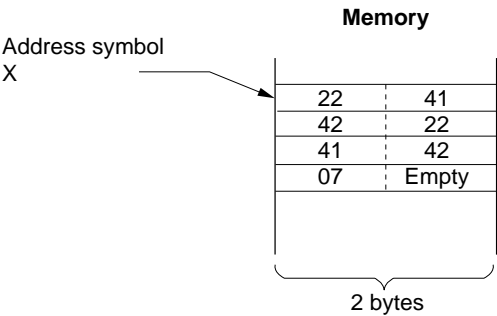
- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Example: ~

```
.SECTION A,DATA,ALGIN=2
.SDATA  ""AB""           ; The string literal in this example
                          ; includes double quotation marks.
.SDATA  "AB"<H'07>       ; The string literal in this example
                          ; has a control code appended.
```

~

Explanatory Figure for the Coding Example



- Notes: 1. The data in this figure is hexadecimal.
2. The ASCII code for “A” is: H'41.  
The ASCII code for “B” is: H'42.  
The ASCII code for “” is: H'22.

## **.SDATAB**

Description Format: [<symbol>[:]]Δ.SDATABΔ<block count>,"<string literal>"

Description: .SDATAB reserves the specified number of strings literal for the block count consecutively in memory.

The <block count> must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

A value of 1 or larger must be specified as the block count.

The maximum value of the block count depends on the length of the string literal data.

The length of the string literal data multiplied by the block count must be less than or equal to H'FFFFFFFF (4,294,967,295 bytes).

When specifying a string literal, enclose the character with double quotation marks (""). When a double quotation mark is used as a character, specify two double quotation marks.

A control code can be appended to a string literal. Enclose the string literal with double quotation marks and then enclose the control code with angle brackets (< >). A control character can be appended to a string literal.

The syntax for this notation is as follows:

"<string literal>"<control code>
----------------------------------

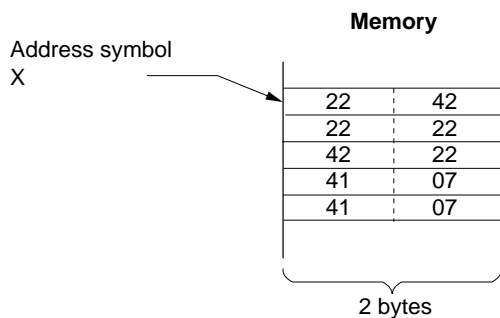
The control code must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Example:

```
~
.SECTION  A,DATA,ALIGN=2
X:
.SDATAB  2,"""B"""" ; The string literal in this
; example includes double quotation
; marks.
.SDATAB  2,"A"<H'07> ; The string literal in this
; example has a control code
; appended.
~
```

### Explanatory Figure for the Coding Example



Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'41.  
The ASCII code for "B" is: H'42.  
The ASCII code for "" is: H'22.

## **.SDATAC**

Description Format: [`<symbol>[:]`]`Δ.SDATACΔ`"`<string literal>`"[,...]

Description: .SDATAC reserves string literal data (with length) in memory. A string literal data with length is reserved with a string literal plus a leading byte that indicates the length of the string. The length indicates the size of the string literal (not including the length) in bytes. When specifying a string literal, enclose the character with double quotation marks ("). When a double quotation mark is used as a character, specify two double quotation marks. A control code can be appended to a string literal. Enclose the string literal with double quotation marks and then enclose the control code with angle brackets (< >).

The syntax for this notation is as follows:

" <code>&lt;string literal&gt;</code> " <code>&lt;control code&gt;</code>
---

The control code must be specified as follows:

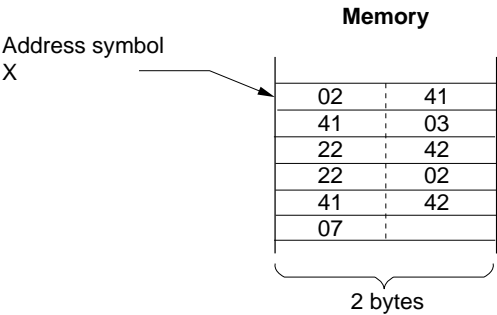
- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Example: ~

```
.SECTION A,DATA,ALGIN=2
X:
.SDATAC "AA" ; This statement reserves character
; string data (with length).
.SDATAC ""B"" ; The string literal in this example
; includes double quotation marks.
.SDATAC "AB"<H'07> ; The string literal in this example
; has a control code appended.
```

~

Explanatory Figure for the Coding Example



- Notes:
- 1. The data in this figure is hexadecimal.
  - 2. The ASCII code for "A" is: H'41.  
The ASCII code for "B" is: H'42.  
The ASCII code for "" is: H'22.

## **.SDATAZ**

Description Format: [<symbol>[:]]Δ.SDATAZΔ"<string literal>"[,...]

Description: .SDATAZ reserves string literal data (with zero terminator) in memory. A string literal with zero terminator is a string literal with an appended trailing byte (with the value H'00) that indicates the end of the string. When specifying a string literal, enclose the character with double quotation marks ("). When a double quotation mark is used as a character, specify two double quotation marks ("). A control code can be appended to a string literal. Enclose the string literal with double quotation marks and then enclose the control code with angle brackets (<>).

The syntax for this notation is as follows:

"<string literal>"<control code>
----------------------------------

The control code must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Example: ~

```
X: .SECTION A,DATA,ALGIN=2
   .SDATAZ "AA" ; This statement reserves character
   .SDATAZ """"B"""; string data (with zero termination).
   .SDATAZ "AB"<H'07> ; The string literal in this example
                        ; includes double quotation marks.
                        ; The string literal in this example
                        ; has a control code appended.
```

~

Explanatory Figure for the Coding Example

Address symbol  
X

41	41
00	22
42	42
22	00
41	42
07	00

2 bytes

Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'41.  
The ASCII code for "B" is: H'42.  
The ASCII code for "" is: H'22.

## .RES

Description Format: [<symbol>[:]]Δ.RES[.<operation size>]Δ<area count>

<operation size> = { B | W | L }

### Description:

.RES reserves data areas in memory.

The integer data of the specified size is reserved for area count.

The operation size determines the size of one area.

The range of values that can be specified as the area count varies with the operation range.

#### Operation Data

Operation Size	Data Size	Area Count Range*
B (byte)	1 byte	H'00000001 to H'FFFFFFFF (1 to 4,294,967,295)
W (word)	2 bytes	H'00000001 to H'7FFFFFFF (1 to 2,147,483,647)
L (longword)	4 bytes	H'00000001 to H'3FFFFFFF (1 to 1,073,741,823)

Note: Numbers in parentheses are decimal.

The byte size is used when the operation size is omitted.

The area count must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.



~

 $\sim$ 

The diagram illustrates memory reservation in a vertical stack of memory units. The top unit is shaded gray and labeled 'Address symbol X' with an arrow. Below it are three white units. The next two units are divided into two columns: the left column is hatched (diagonal lines) and the right column is white. A bracket below these two columns is labeled '2 bytes'. To the right of the diagram, a legend defines the patterns: a gray box for 'Area reserved in longword size.', a white box for 'Area reserved in word size.', and a hatched box for 'Area reserved in byte size.'

## **.SRES**

Description Format: [<symbol>[:]]Δ.SRESΔ<string-literal area size> [...]

Description: .SRES reserves string literal data areas.  
The size of the areas to be reserved is in byte units.  
The string-literal area size must be specified as follows:

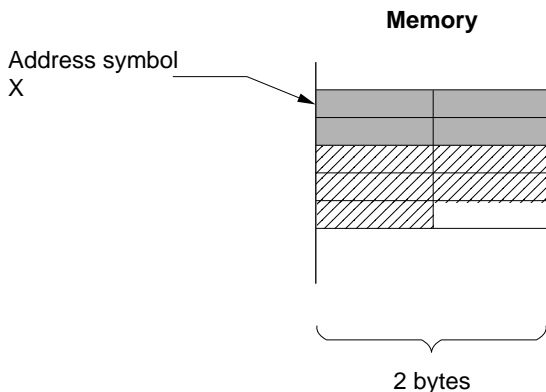
- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

The values that are allowed for the string-literal area size are from H'00000001 to H'FFFFFFFF (from 1 to 4,294,967,295 in decimal).

Example: ~

```
.SECTION A,DATA,ALIGN=2
X:
.SRES    4      ; This statement reserves a 4-byte area.
.SRES    5      ; This statement reserves a 5-byte area
~
```

Explanatory Figure for the Coding Example



## **.SRESC**

Description Format: [<symbol>[:]]Δ.SRESCΔ<string-literal area size>[,...]

Description: .SRESC reserves string literal data areas (with length) in memory. The specified area size (byte count) plus a byte that indicates the length of the string is reserved. The string-literal area size must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

The values that are allowed for the string-literal area size are from H'00000000 to H'000000FF (from 0 to 255 in decimal). The size of the area reserved in memory is the size of the string literal area itself plus 1 byte for the count.

Example:

~

```
.SECTION A,DATA,ALIGN=2
```

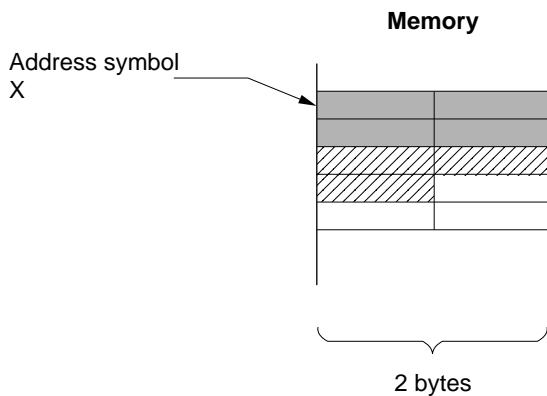
X:

```
.SRESC    3    ; This statement reserves 3 bytes plus 1 byte  
           ; for the count.
```

```
.SRESC    2    ; This statement reserves 2 bytes plus 1 byte  
           ; for the count.
```

~

Explanatory Figure for the Coding Example



## **.SRESZ**

Description Format: [<symbol>[:]]Δ.SRESZΔ<string-literal area size>[,...]

Description: .SRESZ allocates string literal data areas (with zero termination). The specified area size (byte count) plus a byte that indicates zero termination is reserved. The string-literal area size must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

The values that are allowed for the string literal area size are from H'00000000 to H'000000FF (from 0 to 255 in decimal).

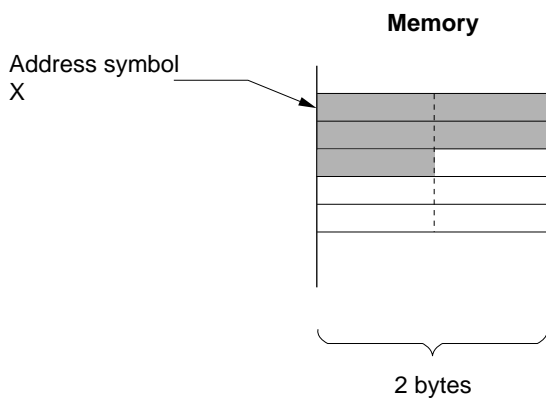
The size of the area reserved in memory is the size of the string literal area itself plus 1 byte for the terminating zero.

Example:

```
.SECTION A,DATA,ALIGN=2  
X:  
  .SRESZ  4 ; This statement reserves 4 bytes plus 1 byte  
           ; for the terminating byte.  
  .SRESZ  3 ; This statement reserves 3 bytes plus 1 byte  
           ; for the terminating byte.
```

~

Explanatory Figure for the Coding Example



## **.EXPORT**

Description Format: `Δ.EXPORTΔ<symbol>[:{8 | 16}][,...]`

The label field is not used.

### **Description:**

`.EXPORT` declares externally defined symbols.

An externally defined symbol declaration is required to reference symbols defined in the current file from other files.

The following can be declared to be externally defined symbols.

- Symbols with an address value
- Absolute address symbols

However, a symbol defined with the `.ASSIGN` directive or a symbol in a dummy section cannot be declared.

By specifying an address size (:8 or :16) with the symbol name, the symbol is addressed in 8- or 16-bit absolute addressing format. Note however that specification of address size for the forward reference symbols is ignored.

Declaration with the `.EXPORT` directive is valid only once in a given program. The assembler ignores the second and later specifications of the `.EXPORT` directive. When the `ABS8` or `NOABS8` directive is specified after this directive is specified, the direction of the `ABS8` or `NOABS8` directive is effective.

To reference a symbol externally from another program source, externally referenced symbols must be declared in the file in which they are referenced using the `.IMPORT` directive according to the externally defined symbol.

Example:

(In this example, a symbol defined in file A is referenced from file B.)

File A:

```
.EXPORT      X      ; This statement declares X to be an  
                  ; externally defined symbol.
```

~

```
X: .EQU      H'10000000 ; This statement defines X.
```

~

File B:

```
.IMPORT      X      ; This statement declares X to be an  
                  ; externally referenced symbol.
```

~

```
.SECTION A,DATA,ALIGN=2  
.DATA.L      X      ; This statement references X.
```

~



## **.IMPORT**

Description Format:  $\Delta$ .IMPORT $\Delta$ <symbol>[: { 8 | 16 }] [...]

The label field is not used.

### **Description:**

.IMPORT declares externally referenced symbols.

An externally referenced symbol declaration is required to reference symbols defined in another source program.

Symbols defined in the current source program cannot be declared to be externally referenced symbols.

By specifying an address size (:8 or :16) with the symbol name, the symbol is addressed in 8- or 16-bit absolute addressing format. Note however that specification of address size for the forward reference symbols is ignored.

Declaration with the .IMPORT directive is valid only once in a given program. The assembler ignores the second and later specifications of the .IMPORT directive. When the ABS8 or NOABS8 directive is specified after this directive is specified, the direction of the ABS8 or NOABS8 directive is effective.

To reference a symbol externally from another program, externally defined symbols must be declared in the file in which they are referenced using the .EXPORT directive.

Example:

(In this example, a symbol defined in file A is referenced from file B.)

**"File A"**

```
.CPU      2600A
.EXPORT   X      ; This statement declares X to be an
                  ; externally defined symbol.
```

~

```
.SECTION  A,CODE,ALIGN=2
X: .EQU    H'10000000 ; This statement defines X.
```

~

**"File B"**

```
.IMPORT   X      ; This statement declares X to be an
                  ; externally referenced symbol.
```

~

```
.SECTION  A,DATA,ALIGN=2
.DATA.L   X      ; This statement references X.
```

~

## **.GLOBAL**

Description Format:  $\Delta$ .GLOBAL $\Delta$ <symbol>[: { 8 | 16 }][,...]

The label field is not used.

### **Description:**

.GLOBAL declares symbols to be either externally defined symbols or externally referenced symbols.

An externally defined symbol declaration is required to reference symbols defined in the source program from other source programs. An externally referenced symbol declaration is required to reference symbols defined in another source program.

A symbol defined within the current source program is declared to be an externally defined symbol by a .GLOBAL declaration.

A symbol that is not defined within the current source program is declared to be an externally referenced symbol by a .GLOBAL declaration.

The following can be declared to be externally defined symbols.

- Symbols with an address value
- Absolute address symbols

However, a symbol defined with the .ASSIGN directive or a symbol in a dummy section cannot be declared.

By specifying an address size (:8 or :16) with the symbol name, the symbol is addressed in 8- or 16-bit absolute addressing format. Note however that specification of address size for the forward reference symbols is ignored.

Declaration with the .GLOBAL directive is valid only once in a given program. The assembler ignores the second and later specifications of the .GLOBAL directive. When the ABS8 or NOABS8 directive is specified after this directive is specified, the direction of the ABS8 or NOABS8 directive is effective.

Example:

```
.CPU      2600A
.GLOBAL  PROG1      ; This statement declares PROG1 to be an
                    ; externally defined symbol.
.GLOBAL  PROG2      ; This statement declares PROG2 to be an
                    ; externally referenced symbol.
:
SECTION   A,CODE,ALIGN=2
PROG1:
MOV.L     ER0,ER1
JSR       @PROG2:24
MOV.L     ER1,ER2
RTS
:
```

## **.BEXPORT**

Description Format:  $\Delta$ .BEXPORT $\Delta$ <symbol>[,...]

The label field is not used.

Description: .BEXPORT declares externally defined symbols for bit data names specified by .BEQU.  
An externally defined symbol declaration is required to reference symbol .BEQU defined in the source program file from other files.

Example: (In this example, a symbol defined in file A is referenced from file B.)  
File A:

```
                .CPU      2600A:12
                .BEXPORT  AD1B0      ; This statement declares AD1B0 to
                                     ; be an externally defined symbol.
AD1             .EQU      H'FFFFFF00
AD1B0           .BEQU     0,AD1
```

~

File B:

```
                .BIMPORT  AD1B0      ; This statement declares AD1B0 to
                                     ; be an externally referenced symbol.
~

                .SECTION  A,CODE,ALIGN=2
                .BSET.B   AD1B0      ; This statement references AD1B0.
```

~

## **.BIMPORT**

Description Format:  $\Delta$ .BIMPORT $\Delta$ <symbol>[,...]

The label field is not used.

Description: .BIMPORT declares externally referenced symbols for bit data names specified by .BEQU.  
When a symbol is defined by a directive other than .BEQU after it is declared by .BIMPORT, a warning occurs. Similarly, when a symbol is declared by .BIMPORT after it is defined by .BEQU, a warning occurs.  
To reference the symbol .BEQU externally from another source program, externally referenced symbols must be declared in the file in which they are referenced using the .BEXPORT directive according to the externally defined symbol.

Example: (In this example, a symbol defined in file A is referenced from file B.)  
File A:

```
.BIMPORT AD1B0 ; This statement declares AD1B0 to be an  
                ; externally referenced symbol.
```

~

```
.SECTION  A,CODE,ALIGN=2  
.BSET.B   AD1B0 ; This statement defines AD1B0.
```

~

File B:

```
.CPU      2600A:32  
.BEXPORT AD1B0 ; This statement declares AD1B0 to be an  
                ; externally defined symbol.  
AD1 .EQU   H'FFFFFF00  
AD1B0 .BEQU 0,AD1
```

~

**.ABS8**  
**.NOABS8**

Description Format:  $\Delta$ .ABS8 $\Delta$ [<symbol>[,...]]

The label field is not used.

$\Delta$ .NOABS8

The label field is not used.

Description: .ABS8 specifies a symbol that is addressed in the 8-bit absolute address format. When only .ABS8 is specified, all the externally referenced or definition symbols after this directive are targeted.  
 When .NOABS8 is specified, all the externally referenced or definition symbols that have been specified with 8-bit absolute address format are excluded from the targets of 8-bit absolute address format after that directive.

Priority of the access size is as follows:

Priority		Format of the Access Size
High	1	Explicitly specified size by the absolute address format
↑	2	Address size specified by the .IMPORT, .EXPORT, and .GLOBAL directives
↓		.ABS8 and .NOABS8 directives
Low	3	abs8 and abs16 options

Example:	.CPU H8SXX:32	
	.IMPORT sym1,sym3,sym5	
	.IMPORT sym2:16	
	.IMPORT sym4:8	
	MOV.B @sym1 ,R1H	;32 bits (no specification)
	MOV.B @sym2 ,R1H	;16 bits (address size specified ; by .IMPORT)
	MOV.B @sym3:8,R1H	;8 bits (explicitly specified size)
	MOV.B @sym4 ,R1H	;8 bits (address size specified ; by .IMPORT)
	MOV.B @sym5 ,R1H	;32 bits (no specification)
	MOV.B @(sym1+sym2),R1H	;16 bits* (no specification and ; 16 bits mixed)
	.ABS8 sym1	
	MOV.B @sym1 ,R1H	;8 bits (.ABS8 specified)
	MOV.B @sym2 ,R1H	;16 bits (address size specified ; by .IMPORT)
	MOV.B @sym3:8,R1H	;8 bits (explicitly specified size)
	MOV.B @sym4 ,R1H	;8 bits (address size specified ; by .IMPORT)
	MOV.B @sym5 ,R1H	;32 bits (no specification)
	MOV.B @(sym1+sym2),R1H	;8 bits* (8 bits and 16 bits ; mixed)
	.NOABS8	
	MOV.B @sym1 ,R1H	;32 bits (.NOABS8 specified)
	MOV.B @sym2 ,R1H	;16 bits (address size specified ; by .IMPORT)
	MOV.B @sym3:8,R1H	;8 bits (explicitly specified size)
	MOV.B @sym4 ,R1H	;32 bits (.NOABS8 specified)
	MOV.B @sym5 ,R1H	;32 bits (no specification)
	MOV.B @(sym1+sym2),R1H	;16 bits* (32 bits and 16 bits ; mixed)

Supplement: When multiple external symbols are described in the absolute address format, the minimum address size is used.



## **.OUTPUT**

Description Format:  $\Delta$ .OUTPUT $\Delta$ <output specifier>[,...]

<output specifier> = { obj | noobj |  
                                  dbg | nodbg }

The label field is not used.

Description: .OUTPUT controls object module and debugging information output.

(1) Output of object module

Controls the output of the object module.

<b>Output Specifier</b>	<b>Output Control</b>
<u>obj</u>	An object module is output.
noobj	No object module is output.

(2) Output of debugging information

Controls the output of the debugging information.

<b>Output Specifier</b>	<b>Output Control</b>
dbg	Debugging information is output in the object module.
<u>nodbg</u>	No debugging information is output in the object module.

If the .OUTPUT directive is used two or more times in a program with inconsistent output specifiers, an error occurs.

The assembler gives priority to command line option specifications concerning the object module and debugging information output.

The default when the output specifier is omitted is obj and nodbg.

(These examples and its description assume that no command line options concerning object module or debugging information output were specified.)

Example 1:       **.OUTPUT**   OBJ               ; An object module is output.  
  ; No debugging information is output.  
  ~

Example 2:       **.OUTPUT**   OBJ,DBG       ; Both an object module and debugging  
  ; information are output.  
  ~

Example 3:       **.OUTPUT**   OBJ,NODBG   ; An object module is output.  
  ; No debugging information is output.  
  ~

Supplement:      Debugging information is required when debugging a program using the  
                    debugger, and is part of the object module.  
                    Debugging information includes information about source statements and  
                    information about symbols.

## **.DEBUG**

Description Format:  $\Delta$ .DEBUG $\Delta$ <output specifier>

<output specifier>= { ON | OFF }

The label field is not used.

Description:

.DEBUG controls the output of symbolic debugging information.

This directive is used to output only those symbols among the symbols in the source program that are necessary for debugging

This directive allows assembly time to be reduced by restricting the output of symbolic debugging information to only those symbols required in debugging.

The specification of the .DEBUG directive is only valid when both an object module and debugging information are output.

<b>Output Specifier</b>	<b>Output Control</b>
<u>on</u>	Symbolic debugging information is output.
off	No symbolic debugging information is output.

The .DEBUG directive can be specified more than once. The specification is valid for the source statement of this directive.

The .DEBUG directive is valid only when the debugging information is output.

The default when the output specifier is omitted is on.

Example:

```
.SECTION A, CODE, ALIGN=2
.DEBUG OFF ; Starting with the next statement, the
           ; assembler does not output symbolic
           ; debugging information.

~

.DEBUG ON ; Starting with the next statement, the
          ; assembler outputs symbolic debugging
          ; information.

~
```

Supplement: The term "symbolic debugging information" refers to the parts of debugging information concerned with symbols.

## .LINE

Description Format:  $\Delta$ .LINE $\Delta$  ["<file name>",]<line number>

The label field is not used.

Description: .LINE changes the file name and line number of the debugging information. The .LINE directive is supported by the C/C++ source level debugging. Accordingly, the .LINE directive is embedded in the assembly source program that is output by the compiler. The file name and the line number managed by the assembler become the values specified by this directive from the next line of the specification. The file name and the line number specified by the .LINE directive are valid only within the specified file.

### Example:

```
ch38 -code=asmcode -debug test.c
```

C source program (test.c)

```
int    func()           /*1*/
{                               /*2*/
    int    i,j;           /*3*/
                               /*4*/
    j=0;                   /*5*/
    for (i=1;i<=10;i++){ /*6*/
        j+=i;             /*7*/
    }                       /*8*/
    return(j);            /*9*/
}
```

→

Assembly source program (test.src)

```
.CPU      2600A:24
.EXPORT   _func
.SECTION  P,CODE,ALIGN=2
.LINE     "/asm/test.c",1
_func:                                ; function: func
.LINE     2
.LINE     5
.LINE     6
SUB.L     ER0,ER0
MOV.B     #1,R0L
.LINE     6
L5:
.LINE     7
ADD.W     R0,E0
.LINE     6
INC.W     #1,R0
.LINE     6
CMP.W     #10,R0
BLE       L5:8
.LINE     8
MOV.W     E0,R0
.LINE     9
RTS
.END
```

## **.DISPSIZE**

Description Format:  $\Delta$ .DISPSIZE $\Delta$ <sub>=<bit count>[,...]

<sub>={ FBR | XBR | FRG | XRG | FWD | XTN | ALL }

The label field is not used.

Description: .DISPSIZE specifies the default size for the displacement of the branch instructions, or when the displacement is the forward reference value or the external reference value for the register indirect with displacement. This directive is available for displacements which has no specification of (:8, :16, :24, :32). <sub> are as follows:

Item	Description
FBR	Forward reference branch instruction
XBR	External reference branch instruction
FRG	Register indirect with forward reference displacement
XRG	Register indirect with external reference displacement
FWD	Specifies FBR and FRG at the same time
XTN	Specifies XBR and XRG at the same time
ALL	Specifies FBR, XBR, FRG, and XRG at the same time

Bit count is as follows:

CPU	Output Method <sup>*1</sup>
H8SX maximum mode	FBR=8, 16, XBR=8, 16, FRG=16, 32, XRG=16, 32, FWD=16, XTN=16, ALL=16
H8SX advanced mode	FBR=8, 16, XBR=8, 16, FRG=16, 32, XRG=16, 32, FWD=16, XTN=16, ALL=16
H8SX middle mode	FBR=8, 16, XBR=8, 16, FRG=16, 32, XRG=16, 32, FWD=16, XTN=16, ALL=16
H8SX normal mode	FBR=8, 16, XBR=8, 16, FRG=16, XRG=16
H8S/2600 advanced mode	FBR=8, <u>16</u> , XBR=8, <u>16</u> , FRG=16, <u>32</u> , XRG=16, <u>32</u> , FWD=16, XTN=16, ALL=16
H8S/2600 normal mode	FBR= <u>8</u> , 16, XBR= <u>8</u> , 16, FRG= <u>16</u> , XRG= <u>16</u>
H8S/2000 advanced mode	FBR=8, <u>16</u> , XBR=8, <u>16</u> , FRG=16, <u>32</u> , XRG=16, <u>32</u> , FWD=16, XTN=16, ALL=16
H8S/2000 normal mode	FBR= <u>8</u> , 16, XBR= <u>8</u> , 16, FRG= <u>16</u> , XRG= <u>16</u>
H8/300H advanced mode	FBR=8, <u>16</u> , XBR=8, <u>16</u> , FRG=16, <u>24</u> , XRG=16, <u>24</u> , FWD=16, XTN=16, ALL=16
H8/300H normal mode	FBR= <u>8</u> , 16, XBR= <u>8</u> , 16, FRG= <u>16</u> , XRG=16
H8/300, H8/300L	FBR= <u>8</u> , XBR= <u>8</u> , FRG= <u>16</u> , XRG= <u>16</u>

Note: Underscored values indicate the settings when specification is omitted.

\*1: In the H8/300 and the H8/300L, FBR=8, XBR=8, FRG=16, and XRG=16 are fixed, so they have no meanings.

The .DISPSIZE directive can be specified more than once.

The specification is valid from the next source statement of this directive.

FBR is valid when the **optimize** option or the **br\_relative** option is not specified.

Example:

```
.CPU      2600A
.SECTION  A,DATA,ALIGN=2

.DISPSIZE FBR=16          ; [1]
BRA      sym              ; Same as BRA sym:16

.DISPSIZE FBR=8          ; [2]
BRA      sym              ; Same as BRA sym:8
sym:
MOV.W    R0,R1
```

[1]: Sets the displacement size of the forward reference branch instruction to 16 bits.

[2]: Sets the displacement size of the forward reference branch instruction to 8 bits.



## .PRINT

Description Format: Δ.PRINTΔ<output specifier>[,...]

<output specifier>={ LIST | NOLIST | SRC | NOSRC |  
CREF | NOCREF | SCT | NOSCT }

The label field is not used.

Description: .PRINT controls the following output.

- (1) Assemble listing
- (2) Source program listing
- (3) Cross-reference listing
- (4) Section information listing

Item	Output Specifier* <sup>1</sup>	Assembler Action
(1)	list	An assemble listing is output.* <sup>2</sup>
	<u>nolist</u>	No assemble listing is output.* <sup>2</sup>
(2)	<u>src</u>	A source program listing is output in the assemble listing.* <sup>3*4</sup>
	nosrc	No source program listing is output in the assemble listing.* <sup>3*4</sup>
(3)	<u>cref</u>	A cross-reference listing is output in the assemble listing.* <sup>3*5</sup>
	nocref	No cross-reference listing is output in the assemble listing.* <sup>3*5</sup>
(4)	<u>sct</u>	A section information listing is output in the assemble listing.* <sup>3*6</sup>
	nosct	No section information listing is output in the assemble listing.* <sup>3*6</sup>

- Notes:
1. This specification is valid only once.
  2. Valid when the **list** or **nolist** option is not specified.
  3. Valid when the assemble listing is output.
  4. Valid when the **source** or **nosource** option is not specified.
  5. Valid when the **cross\_reference** or **nocross\_reference** option is not specified.
  6. Valid when the **section** or **nosection** option is not specified.

If the .PRINT directive is used two or more times in a program with inconsistent output specifiers, an error occurs.

```
Example:      .PRINT      LIST,SRC,NOCREF,NOSCT
              ;
              .SECTION    A,CODE,ALIGN=2
START
              MOV.W       R0,R1
              MOV.W       R0,R2
```

Only a source program listing is output in the assemble listing.

## .LIST

Description Format:  $\Delta$ .LIST $\Delta$ <output specifier>[,...]

$\Delta$ <output specifier>={ ON | OFF | COND | NOCOND | DEF | NODEF |  
CALL | NOCALL | EXP | NOEXP | STR |  
NOSTR | CODE | NOCODE }

The label field is not used.

Description: .LIST controls output of the source program listing in the following three ways:

- (1) Selects whether or not to output source statements.
- (2) Selects whether or not to output source statements related to the preprocessor function.
- (3) Selects whether or not to output object code lines.

Output is controlled by output specifiers as follows:

Output Specifier				
Type	Output	Not output	Object	Description
a	<u>on</u>	off	Source statements	The source statements following this directive
b	<u>cond</u>	nocond	Failed condition*	Condition-failed .AIF or .AIFDEF directive statements
	<u>def</u>	nodef	Definition*	Macro definition statements .AREPEAT and .AWHILE definition statements .INCLUDE directive statements .ASSIGNA and .ASSIGNC directive statements
	<u>call</u>	nocall	Call*	Macro call statements, .AIF, AIFDEF, and .AENDI directive statements
	<u>exp</u>	noexp	Expansion*	Macro expansion statements .AREPEAT and .AWHILE expansion statements
	<u>str</u>	nostr	Structured assembly*	Structured assembly expansion statements
c	<u>code</u>	nocode	Object code lines*	The object code lines exceeding the source statement lines

Note: This specification is valid when the **show** or **noshow** option is not specified.

.LIST directive statements themselves are not output on the source program listing.

The specification of the .LIST directive is only valid for the source statements after the specification.

Example:

```

                                .PRINT      list
;
                                .list       off                ; [1]
                                .include     "bbb.h"            ;
                                .list       on                  ;[2]
                                .section     A,CODE,ALIGN=2
START
                                MOV.W       R0,R1
                                MOV.W       R0,R2
```

The .LIST directive suppresses the output of part of the source statements. Source statements between [1] and [2] are not output to the source program listing.

**.FORM**

Description Format: Δ.FORMΔ<size specifier>[,...]

<size specifier> = { LIN = <line count> | COL = <column count> }

The label field is not used.

Description: .FORM sets the number of lines per page and columns per line in the assemble listing.

The line count and column count must be specified as follows:

- The specifications must be constant values, and,
- Forward reference symbols must not appear in the specifications.

Size Specifier	Listing Size	Allowable Range* <sup>3</sup>	When Not Specified
LIN=<line count>	The specified value is set to the number of lines per page.* <sup>1</sup>	20 to 255	60
COL= <column count>	The specified value is set to the number of columns per line.* <sup>2</sup>	79 to 255	132

Notes: 1. Valid when the lines option is not specified.  
2. Valid when the columns option is not specified.  
3. When a value less than 20 is specified, 20 is assumed, and when a value more than 255 is specified, 255 is assumed, and no error is output.

The assembler gives priority to command line option specifications concerning the number of lines and columns in the assemble listing. The .FORM directive can be used any number of times in a given source program. The specification of size becomes valid starting from the next page of this directive.

Example:

~

**.FORM** LIN=60,COL=200

; Starting with this page, the number of  
; lines  
; per page in the assemble listing is 60  
; lines.  
; Also, starting with this line, the number  
; of columns per line in the assemble  
; listing is 200 columns.

~

**.FORM** LIN=55,COL=150

; Starting with this page, the number of  
; lines  
; per page in the assemble listing is 55  
; lines.  
; Also, starting with this line, the number  
; of columns per line in the assemble  
; listing is 150 columns.

~

## **.HEADING**

Description Format:  $\Delta$ .HEADING $\Delta$ "<string literal>"

The label field is not used.

Description: .HEADING sets the title in the header for the source program listing. A string literal of up to 60 characters can be specified as the header. Even when the number of characters exceeds 60 characters, no error message is output.

When specifying a string literal, enclose the character with double quotation marks (""). When a double quotation mark is used as a character, specify two double quotation marks.

The range of validity for a given use of the .HEADING directive is as follows:

- When the .HEADING directive is on the first line of a page, it is valid starting with that page.
- When the .HEADING directive appears on the second or later line of a page, it is valid starting with the next page.

The .HEADING directive can be used any number of times in a given source program.

Example:

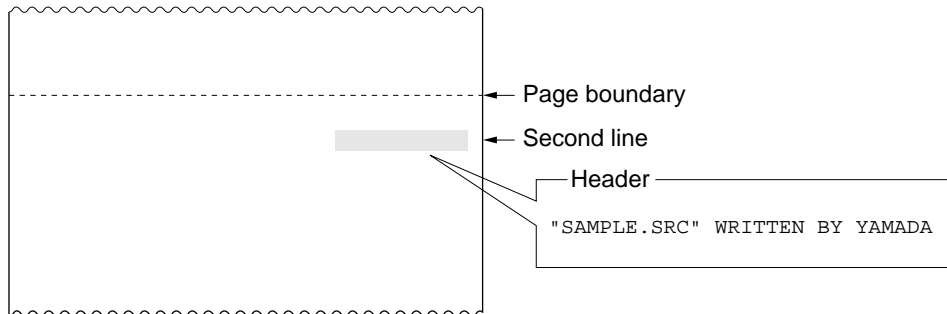
~

**.HEADING** ""SAMPLE.SRC"" WRITTEN BY YAMADA"

~

### Explanatory Figure for the Coding Example

#### Source program listing





## **.PAGE**

Description Format: Δ.PAGE

The label field is not used.

Description: .PAGE inserts a new page in the source program listing.  
The .PAGE directive is ignored if it is used on the first line of a page.  
.PAGE directive statements themselves are not output to the source program listing.  
This directive is valid when the source program listing is output.

Example:     ~

```
                .PRINT    LIST
                .SECTION  A, CODE, ALIGN=2
START
                .MOV.W    R0, R1
                .MOV.W    R0, R2
                ;
                .PAGE
                .SECTION  B, DATA, ALIGN=2
DAT
                .DATA.W   H'0001
                .DATA.W   H'0002
                ~
```

## Explanatory Figure for the Coding Example

### Source program listing

```
4 00000000 0D01          4          MOV.W    R0,R1
5 00000022 0D02          5          MOV.W    R0,R2
```

```
*** H8S,H8/300 ASSEMBLER Ver. 4.0 ***      07/18/00 21:28:14
PROGRAM NAME =
```

```
9 00000000          9          .SECTION  B,DATA,ALIGN=2
10 00000000          10 DAT
11 00000000 0001      11          .DATA.W   H'0001
12 00000002 0002      12          .DATA.W   H'0002
```

← New  
page

## **.SPACE**

Description Format:  $\Delta$ .SPACE[ $\Delta$ <line count>]

The label field is not used.

Description: .SPACE outputs the specified number of blank lines to the source program listing. A single blank line is output if this operand is omitted.  
The line count must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Values from 1 to 50 can be specified as the line count.

If a value less than 1 is specified, 1 is assumed. If a value more than 50 is specified, 50 is assumed. In these cases, no error message is output.

Nothing is output for the lines output by the .SPACE directive; in particular line numbers are not output for these lines.

When a new page occurs as the result of blank lines output by the .SPACE directive, any remaining blank lines are not output on the new page.

.SPACE directive statements themselves are not output to the source program listing.

This directive is valid when the source program listing is output.

Example:

```
.SECTION  A,DATA,ALIGN=2
.DATA.W   H'1111
.DATA.W   H'2222
.DATA.W   H'3333
.DATA.W   H'4444           ;Inserts five blank lines at the point
                           ; where the section changes.
.SPACE 5
.SECTION  B,DATA,ALIGN=2
~
```

## Explanatory Figure for the Coding Example

### Source program listing

```
*** H8S, H8/300 ASSEMBLER Ver. 4.0 ***      07/18/00 13:35:58
PROGRAM NAME =

1  00000000          1          .SECTION   A,DATA,ALIGN=2
2  00000000  1111      2          .DATA.L   H'1111
3  00000002  2222      3          .DATA.L   H'2222
4  00000004  3333      4          .DATA.L   H'3333
5  00000006  4444      5          .DATA.L   H'4444

7                                7          .SECTION   B,DATA,ALIGN=2
```

## **.PROGRAM**

Description Format: **Δ.PROGRAMΔ**<object module name>

The label field is not used.

### Description:

**.PROGRAM** sets the object module name.

The object module name is a name that is required by the optimizing linkage editor to identify the object module.

Object module naming conventions are the same as symbol naming conventions.

The assembler distinguishes uppercase and lowercase letter in object module names.

Setting the object module name with the **.PROGRAM** directive is valid only once in a given program. The assembler ignores the second and later specifications of the **.PROGRAM** directive.

If there is no **.PROGRAM** specification of the object module name, the assembler will set a default (implicit) object module name.

The default object module name is the file name of the object file (the object module output destination).

Example: Object file name ..... **PROG**     **.obj**  
  ||                    ||  
  File name     File type  
  ↓  
Object module name ..... **PROG**

The object module name can be the same as a symbol used in the program.

Example: **.PROGRAM** **PROG1**     ; This statement sets the object module  
  ; name to be **PROG1**.

~

## **.RADIX**

Description Format:  $\Delta$ .RADIX $\Delta$ <radix specifier>

<radix specifier> = { B | Q | D | H }

The label field is not used.

Description:

.RADIX sets the radix (base) for integer constants with no radix specification.

This specifier sets the radix (base) for integer constants with no radix specification.

If hexadecimal (radix specifier H) is specified as the radix for integer constants with no radix specification, integer constants whose first digit is A through F must be prefixed with a 0 (zero). (The assembler interprets expressions that begin with A through F to be symbols.)

Specifications with the .RADIX directive are valid from the point of specification forward in the program.

<b>Radix Specifier</b>	<b>Integer Constant with no Radix</b>
B	Binary
Q	Octal
<u>D</u>	Decimal
H	Hexadecimal

When there is no radix specification with the .RADIX directive in a program, integer constants with no radix specification are interpreted as decimal constants.

Example: 1.

```
~  
X:  .RADIX D  
    .EQU      100 ;This 100 is decimal.
```

```
~  
Y:  .RADIX H  
    .EQU      64 ;This 64 is hexadecimal.
```

2.

```
~  
Z:  .RADIX H  
    .EQU      0F ; A zero is prefixed to this constant "0F" since it  
                ; would be interpreted as a symbol if it were  
                ; written as simply "F".  
~
```

**.END**

Description Format: Δ.ENDΔ<symbol>

The label field is not used.

Description: .END sets the end of the source program and the entry point.  
The assembly processing ends when the .END directive is detected.  
A symbol specified for an operand is regarded as the entry point.  
An externally defined symbol is specified for the symbol.

Example:

```
      .EXPORT  START  
      .SECTION P,CODE,ALIGN=2  
START:  
~  
      .END      START ;Declares the end of the source program.  
                  ;Symbol START becomes the entry point.
```

## **.STACK**

Description Format:  $\Delta$ .STACK $\Delta$ <symbol> = <stack value>

The label field is not used.

Description: .STACK defines the stack amount for a specified symbol referenced by using the stack analysis tool.

The stack value for a symbol can be defined only one time; the second and later specifications for the same symbol are ignored. A multiple of 2 in the range from H'00000000 to H'FFFFFFFE can be specified for the stack value, and any other value is invalid.

The stack value must be specified as follows:

- A constant value must be specified.
- Forward reference symbol, external reference symbol, and relative address symbol must not be used.

Example:

```
~  
.STACK    SYMBOL=H'100  
~
```



# 11.4 File Inclusion Function

The file inclusion function allows source files to be included into other source files. The file included into another file is called an include file.

This assembler provides the `.INCLUDE` directive to perform file inclusion.

The file specified with the `.INCLUDE` directive is inserted at the location of the `.INCLUDE` directive.

Example:

Source program

`.INCLUDE "FILE.H"`

`.SECTION CD1, CODE, ALIGN=2`

`MOV #ON, R0`

~

Include file FILE.H

`ON: .EQU 1`

`OFF: .EQU 0`



File included result (source list)

`.INCLUDE "FILE.H"`

`ON: .EQU 1`

`OFF: .EQU 0`

`.SECTION CD1, CODE, ALIGN=2`

`MOV #ON, R0`

~

## .INCLUDE

Description Format: `Δ.INCLUDEΔ"<file name>"`

The label field is not used.

### Description:

.INCLUDE is the file inclusion assembler directive. If no file extension is specified, only the file name is used as specified (the assembler does not assume any default file extension).

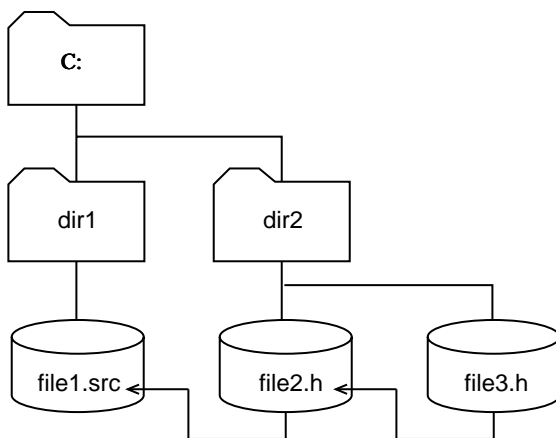
The file name can include the directory path name. The directory can be specified either by the absolute path (path from the root directory) or by the relative path (path from the current directory).

Included files can include other files. The nesting depth for file inclusion is limited to 30 levels.

The directory name of the filenames specified by .INCLUDE can be changed by the **include** option.

### Example:

This example assumes the following directory configuration and operations:



- Starts the assembler from the root directory (c:\)
- Inputs source file c:\dir1\file1.src
- Makes file2.h included in file1.src
- Makes file3.h included in file2.h

The start command is as follows:

```
>asm38 c:\dir1\file1.src (RET)
```

file1.src must have the following include directive:

```

        .INCLUDE "dir2\file2.h"      ;      \ is the current directory
                                         ;      (relative path
                                         ;      specification).
or
        .INCLUDE "\dir2\file2.h"    ;      Absolute path
                                         ;      specification

```

file2.h must have the following inclusion directive:

```

        .INCLUDE "file3.h"          ;      \dir2 is the current directory
                                         ;      (relative path specification).
or
        .INCLUDE "\dir2\file3.h"    ;      Absolute path
                                         ;      specification

```

## Notes

When using UNIX, change the backslash (\) in the above example to slash (/).

## 11.5 Conditional Assembly Function

### 11.5.1 Overview of the Conditional Assembly Function

The conditional assembly function provides the following assembly operations:

- Replaces a string literal in the source program with another string literal.
- Selects whether or not to assemble a specified part of a source program according to the condition.
- Iteratively assembles a specified part of a source program.

#### (1) Preprocessor variables

Preprocessor variables are used to write assembly conditions. Preprocessor variables are of either integer or character type.

##### (a) Integer preprocessor variables

Integer preprocessor variables are defined by the `.ASSIGNA` directive or the **assigna** option (these variables can be redefined by the `.ASSIGNA` directive).

When referencing integer preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

A coding example is shown below:

Example:

```
FLAG: .ASSIGNA 1                ; An integer value of 1 is set to FLAG.
      ~
      .AIF \&FLAG EQ 1          ; MOV R0,R1 is assembled.
      MOV.W R0,R1               ; when FLAG is 1.
      .AENDI
      ~
```

##### (b) Character preprocessor variables

Character preprocessor variables are defined by the `.ASSIGNC` directive or the **assignc** option (these variables can be redefined by the `.ASSIGNC` directive).

When referencing character preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

A coding example is shown below:

Example:

```
FLAG:  .ASSIGNC "ON"           ; String literal ON is set to FLAG.
      ~
      .AIF "&FLAG" EQ "ON"    ; MOV.W R0,R1 is assembled
      MOV.W R0,R1             ; when FLAG is "ON".
      .AENDI
      ~
```

## (2) Replacement Symbols

The .DEFINE directive specifies symbols that will be replaced with the corresponding string literals at assembly. A coding example is shown below.

Example:

```
SYM1:  .DEFINE      "R1"
      ~
      MOV.W      SYM1,R0    ; Replaced with MOV.W R1,R0.
      ~
```

### (3) Conditional Assembly

The conditional assembly function determines whether or not to assemble a specified part of a source program according to the (specified) conditions. Conditional assembly is classified into two types: conditional assembly with comparison using relational operators and conditional assembly with definition of replacement symbols.

#### (a) Conditional Assembly with Comparison

Selects the part of program to be assembled according to whether or not the specified condition is satisfied. A coding example is as follows:

```

~
.AIF <comparison condition 1>
    <Statements to be assembled when condition 1 is satisfied>
.AELIF <comparison condition 2>
    <Statements to be assembled when condition 2 is satisfied>
.AELSE
    <Statements to be assembled when both conditions are not satisfied>
.AENDI
~

```

---This part can be omitted.

Example:

```

~
.AIF "&FLAG" EQ "ON"
    MOV.W R0,R02          ; Assembled when FLAG
    MOV.W R1,R3           ; is ON.
    MOV.W R2,R0           ;
.AELSE
    MOV.W R2,R0           ; Assembled when FLAG
    MOV.W R3,R1           ; is not ON.
.AENDI
~

```

## (b) Conditional Assembly with Definition

Selects the part of program to be assembled by whether or not the specified replacement symbol has been defined. A coding example is as follows:

```

~
.AIFDEF  <definition condition>
  <Statements to be assembled when the specified replacement symbol is defined>
~-----~
.AELSE
  <Statements to be assembled when the specified replacement symbol is not defined>
~-----~
.AENDI
~

```

---This part can be omitted.

Example:

```

~
.AIFDEF  FLAG
  MOV.W  R0,R3          ; Assembled when FLAG is defined with
  MOV.W  R1,R4          ; the .DEFINE directive before the .AIFDEF
  MOV.W  R2,R5          ; directive in the program.
.AELSE
  MOV.W  R3,R0          ; Assembled when FLAG is not defined with
  MOV.W  R4,R1          ; the .DEFINE directive before the .AIFDEF
  MOV.W  R5,R2          ; directive in the program.
.AENDI
~

```

#### (4) Iterated Expansion

A part of a source program can be iteratively assembled the specified number of times. A coding example is shown below.

Example:

```

~
.AREPEAT <count>
    <Statements to be iterated>
.AENDR
~

```

Example:

```

MOV.B    R1L,R1H
.AREPEAT  2          ; The iterated count is specified.
ADD.B    R0L,R1L
ADD.B    R2L,R3L
.AENDR

```

After expansion

```

MOV.B    R1L,R1H
ADD.B    R0L,R1L
ADD.B    R2L,R3L
ADD.B    R0L,R1L
ADD.B    R2L,R3L
          } Expanded part
ADD.B    R3L,R1L

```

Source statements between .AREPEAT and .AENDR are iterated twice by expansion, and are assembled.



## (5) Conditional Iterated Expansion

A part of a source program can be iteratively assembled while the specified condition is satisfied. A coding example is shown below.

```

~
.AWHILE <condition>
    <Statements to be iterated>
.AENDW
~

```

Example:

```

COUNT    .ASSIGNA    2                                ; The iterated count is specified.
           .AWHILE     \&COUNT NE 0                    ; Expanded while COUNT is not 0.
           ADD.B       R0L,R1L
           ADD.B       R0L,R2L
           INC.B       R0L
COUNT     .ASSIGNA     \&COUNT-1                      ; COUNT minus 1.
           .AENDW
           MOV.B       R0L,@SP

```

After expansion

```

           MOV.B       R0H,R0L
           ADD.B       R0L,R1L
           ADD.B       R2L,R3L
           INC.B       R0L
COUNT     .ASSIGNA     \&COUNT-1
           ADD.B       R0L,R1L
           ADD.B       R0L,R2L
           INC.B       R0L
COUNT     .ASSIGNA     \&COUNT-1
           MOV.B       R0L,@SP

```

} Expanded part

Source statements between .AWHILE and .AENDW are iterated while COUNT is not zero by expansion, and are assembled.

### 11.5.2 Conditional Assembly Directives

This assembler provides the conditional assembly directives shown in table 11.15.

**Table 11.15 Conditional Assembly Directives**

Category	Mnemonic	Function
Variable definition	.ASSIGNA	Defines an integer preprocessor variable. The defined variable can be redefined.
	.ASSIGNC	Defines a character preprocessor variable. The defined variable can be redefined.
	.DEFINE	Defines a preprocessor replacement string literal.
Conditional branch	.AIF	Determines whether or not to assemble a part of a source program according to the specified condition. When the condition is satisfied, the statements after the .AIF are assembled. When not satisfied, the statements after the .AELIF or .AELSE are assembled.
	.AELIF	
	.AELSE	
	.AENDI	
	.AIFDEF	Determines whether or not to assemble a part of a source program according to the replacement symbol definition. When the replacement symbol is defined, the statements after the .AIFDEF are assembled. When not defined, the statements after the .AELSE are assembled.
	.AELSE	
	.AENDI	
Iterated expansion	.AREPEAT	Repeats assembly of a part of a source program (between .AREPEAT and .AENDR) the specified number of times.
	.AENDR	
	.AWHILE	Assembles a part of a source program (between .AWHILE and .AENDW) iteratively while the specified condition is satisfied.
	.AENDW	
Others	.EXITM	Terminates .AREPEAT or .AWHILE iterated expansion.
	.AERROR	Performs error processing in preprocessor expansion.
	.ALIMIT	Specifies the maximum count of .AWHILE expansion.

## .ASSIGNA

Description Format: <preprocessor variable>[:] $\Delta$ . ASSIGNA  $\Delta$  <value>

Description: .ASSIGNA defines a value for an integer preprocessor variable. The syntax of integer preprocessor variables is the same as that for symbols. An integer preprocessor variable can be defined with up to 32 characters, and uppercase and lowercase letters are distinguished.

The preprocessor variables defined with the .ASSIGNA directive can be redefined with the .ASSIGNA directive.

The value to be assigned has the following format:

- Constant (integer constant and character constant)
- Defined preprocessor variable
- Expression using the above as terms

Defined preprocessor variables are valid in the source statements following the directive.

Defined preprocessor variables can be referenced in the following locations:

- .ASSIGNA directive
- .ASSIGNC directive
- .AIF directive
- .AELIF directive
- .AREPEAT directive
- .AWHILE directive
- Macro body (source statements between .MACRO and .ENDM)

When referencing integer preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

`\&<preprocessor variable>[ ' ]`

To clearly distinguish the preprocessor variable name from the rest of the source statement, an apostrophe (') can be added.

When a preprocessor string literal is defined by a command line option, the .ASSIGNA directive specifying the preprocessor variable having the same name as the string literal is invalidated.

### Example:

```
FLAG    .ASSIGNA  1          ; FLAG is set to 1.
;
        .SECTION  A, CODE, ALIGN=2
START
        .AIF \&FLAG EQ 1      ; Same as .AIF 1 EQ 1.
        MOV.W   R0,R2
        .AENDI
        .AIF \&FLAG EQ 2      ; Same as .AIF 1 EQ 2.
        MOV.W   R1,R2
        .AENDI
;
FLAG    .ASSIGNA  2          ; FLAG is changed to 2.
;
        .AIF \&FLAG EQ 1      ; Same as .AIF 1 EQ 1.
        MOV.W   R0,R2
        .AENDI
        .AIF \&FLAG EQ 2      ; Same as .AIF 1 EQ 2.
        MOV.W   R1,R2
        .AENDI
```

Integer preprocessor variable FLAG is referenced by .AIF.

## **.ASSIGNC**

Description Format: <preprocessor variable>[:] $\Delta$ .ASSIGNC $\Delta$ "<string literal>"

Description: .ASSIGNC defines a string literal for a character preprocessor variable. The syntax of character preprocessor variables is the same as that for symbols. A character preprocessor variable can be defined with up to 32 characters, and uppercase and lowercase letters are distinguished. The preprocessor variables defined with the .ASSIGNC directive can be redefined with the .ASSIGNC directive.

String literals are specified by characters or preprocessor variables enclosed with double quotation marks (").

Defined preprocessor variables are valid in the source statements following the directive.

Defined preprocessor variables can be referenced in the following locations:

- .ASSIGNA directive
- .ASSIGNC directive
- .AIF directive
- .AELIF directive
- .AREPEAT directive
- .AWHILE directive
- Macro body (source statements between .MACRO and .ENDM)

When referencing character preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

`\&<preprocessor variable>[ ' ]`

To clearly distinguish the preprocessor variable name from the rest of the source statement, an apostrophe (') can be added.

When a preprocessor string literal is defined by a command line option, the .ASSIGNC directive specifying the preprocessor variable having the same name as the string literal is invalidated.

### Example:

```
FLAG1  .ASSIGNC  "ON"                ; FLAG1 is set to ON.
;
      .SECTION  A, CODE, ALIGN=2
START
      .AIF  "\&FLAG1" EQ "ON"        ; Same as .AIF "ON" EQ "ON".
      MOV.W  R0, R2
      .AENDI
      .AIF  "\&FLAG1" EQ "OFF"       ; Same as .AIF "ON" EQ "OFF".
;
FLAG2  .ASSIGNC  "OFF"               ; FLAG is changed to string literal OFF.
;
      .AIF  "\&FLAG2" EQ "ON"        ; Same as .AIF "ON" EQ "ON"
      MOV.W  R3, R5
      .AENDI
      .AIF  "\&FLAG2" EQ "OFF"       ; Same as .AIF "OFF" EQ "OFF".
      MOV.W  R4, R5
      .AENDI
FLAG   .ASSIGNC  "\&FLAG1' \&FLAG2"  ; "" is used to distinguish between FLAG and AND.
                                           ; FLAG becomes "ON AND OFF" as a result.
```

Character preprocessor variable FLAG is referenced by .AIF.

## .DEFINE

Description Format: <symbol>[:] $\Delta$ .DEFINE $\Delta$ "<replacement string literal>"

Description: .DEFINE specifies that the symbol is replaced with the corresponding string literal.

The differences between the .DEFINE directive and the .ASSIGNC directive are as follows.

- The symbol defined by the .ASSIGNC directive can only be used in the preprocessor statement; the symbol defined by the .DEFINE directive can be used in any statement.
- The symbols defined by the .ASSIGNA and the .ASSIGNC directives are referenced by the "&symbol" format; the symbol defined by the .DEFINE directive is referenced by the "symbol" format.
- The .DEFINE symbol cannot be redefined.
- The .DEFINE directive specifying a symbol is invalidated when the same replacement symbol has been defined by a command line option.

Example:

```
SYM1: .DEFINE      "R1 "
```

~

```
MOV.W      SYM1,R0      ; Replaced with MOV.W  R1,R0.
```

~

A hexadecimal number starting with an alphabetical character a to f or A to F will be replaced when the same string literal is specified as a replacement symbol by the .DEFINE directive. Add 0 to the beginning of the number to stop replacing such number.

```
A0: .DEFINE      "0 "
```

```
MOV.W      #H' A0 ,R0      ; Replaced with MOV.B #H'0,R0.
```

```
MOV.W      #H' 0A0 ,R0     ; Not replaced.
```

A radix indication (B', Q', D', or H') will also be replaced when the same string literal is specified as a replacement symbol by .DEFINE directive. When specifying a symbol having only one character, such as B, Q, D, H, b, q, d, or h, make sure that the corresponding radix indication is not used.

```
B: .DEFINE      "H "
```

```
MOV.W      #B' 10 ,R0      ; Replaced with MOV.W #H'10,R0.
```

Remarks: The replacement is not applied to the **.AENDI**, **.AENDR**, **.AENDW**, **.AIFDEF**, **.END**, **.ENDM**, **.ENDF**, **.ENDI**, **.ENDS**, and **.ENDW** directives.

## **.AIF, .AELIF, .AELSE, .AENDI**

Description Format:  $\Delta$ .AIF $\Delta$ <term1> $\Delta$ <relational operator> $\Delta$ <term2>  
<Source statements assembled if the AIF condition is satisfied>  
[ $\Delta$ .AELIF $\Delta$ <term1> $\Delta$ <relational operator> $\Delta$ <term2>  
<Source statements assembled if the AELIF condition is satisfied>]  
[ $\Delta$ .AELSE  
<Source statements assembled if all the conditions are not satisfied>]  
.AENDI

The label field is not used.

Description: .AIF, .AELIF, .AELSE, and .AENDI select whether or not to assemble source statements according to the condition specified. The .AELIF and .AELSE directives can be omitted.  
.AELIF can be specified repeatedly between .AIF and .AELSE.  
The operand must be specified as follows:  
.AIF: Condition to be compared.  
.AELIF: Condition to be compared.  
.ALESE: Operand field cannot be used.  
.AENDI: Operand field cannot be used.  
Terms are specified with numeric values or string literals. However, when a numeric value and a string literal are compared, the condition always fails.  
Numeric values are specified by constants or preprocessor variables.  
String literals are specified by characters or preprocessor variables enclosed with double quotation marks ("). When a double quotation mark is used as a character, specify two double quotation marks.  
The following relational operators can be used:  
EQ: term1 = term2  
NE: term1  $\neq$  term2  
GT: term1 > term2  
LT: term1 < term2  
GE: term1  $\geq$  term2  
LE: term1  $\leq$  term2

Note: For string literals, only EQ and NE conditions can be used.



Example:

~

```
.AIF \&TYPE EQ 1
MOV.W  R0,R3      ; These statements
MOV.W  R1,R4      ; are assembled when TYPE is 1.
.AELIF \&TYPE EQ 2
MOV.W  R0,R2      ; These statements
MOV.W  R1,R3      ; are assembled when TYPE is 2.
.AELSE
MOV.W  R0,R4      ; These statements
MOV.W  R1,R5      ; are assembled when TYPE is not 1 nor 2.
.AENDI
```

~

## **.AIFDEF, .AELSE, .AENDI**

Description Format:  $\Delta$ .AIFDEF $\Delta$ <replacement symbol>

<statements to be assembled when the specified replacement symbol is defined>

[ $\Delta$ .AELSE

<statements to be assembled when the specified replacement symbol is not defined>]

.AENDI

The label field is not used.

Operation: Enter the .AIFDEF, .AELSE (can be omitted), or .AENDI.

Description:

.AIFDEF, .AELSE, and .AENDI select whether or not to assemble source statements according to the replacement symbol definition.

.AELSE can be omitted.

The operand must be specified as follows:

.AIFDEF: The condition to be defined.

.AELSE: The operand field cannot be used.

.AENDI: The operand field cannot be used.

The replacement symbol can be defined by the .DEFINE directive or the **define** option.

When the specified replacement symbol is defined by the command line option or defined before being referenced by these directives, the condition is regarded as satisfied. When the replacement symbol is defined after being referenced by these directives or is not defined, the condition is regarded as unsatisfied.

Example:

~

```
.AIFDEF      FLAG
MOV.W        R0,R3      ; These statements are assembled when
MOV.W        R1,R4      ; FLAG is defined by the .DEFINE directive.
.AELSE
MOV.W        R0,R2      ; These statements are assembled when
MOV.W        R1,R3      ; FLAG is not defined by the .DEFINE directive.
.AENDI
```

~

## **.AREPEAT, .AENDR**

Description Format:  $\Delta$ .AREPEAT $\Delta$ <count> <source statements iteratively assembled>  
 $\Delta$ .AENDR

The label field is not used.

Description: .AREPEAT and .AENDR assemble source statements by iteratively expanding them the specified number of times.

The operand must be specified as follows.

.AREPEAT: The number of iterations.

.AENDR: The operand field cannot be used.

The source statements between the .AREPEAT and .AENDR directives are iterated the number of times specified with the .AREPEAT directive. (Note that the source statements are simply copied the specified number of times, and therefore, the operation is not a loop at program execution.)

Counts are specified by constants or preprocessor variables.

Nothing is expanded if a value of 0 or smaller is specified.

Example:

```
MOV.B    @SP,R0L
.AREPEAT 3
SHAL.B   R0L
.AENDR
MOV.B    R0L,@SP
```

Expanded results are as follows:

```
MOV.B    @SP,R0L
SHAL.B   R0L
SHAL.B   R0L
SHAL.B   R0L
MOV.B    R0L,@SP
```

## **.AWHILE, .AENDW**

Description Format:  $\Delta$ .AWHILE $\Delta$ <term1> $\Delta$ <relational operator> $\Delta$ <term2>  
<Source statements iteratively assembled>  
 $\Delta$ .AENDW

The label field is not used.

Description: .AWHILE and .AENDW assemble source statements by iteratively expanding them while the specified condition is satisfied.  
The operand must be specified as follows.  
.AWHILE: The condition to iteratively expand source statements.  
.AENDW: The operand field cannot be used.

The source statements between the .AWHILE and .AENDW directives are iterated while the condition specified with the .AWHILE directive is satisfied. Note that the source statements are simply copied iteratively, and therefore, the operation is not a loop at program execution.

Terms are specified with numeric values or string literals. However, when a numeric value and a string literal are compared, the condition always fails.

Numeric values are specified by constants or preprocessor variables.

String literals are specified by characters or preprocessor variables enclosed with double quotation marks (""). When a double quotation mark is used as a character, specify two double quotation marks.

Conditional iterated expansion terminates when the condition finally fails.

If a condition which never fails is specified, source statements are iteratively expanded for 65,535 times or until the maximum count of statement expansion specified by the .ALIMIT directive is reached. Accordingly, the condition for this directive must be carefully specified.

The following relational operators can be used:

EQ: term1 = term2

NE: term1  $\neq$  term2

GT: term1 > term2

LT: term1 < term2

GE: term1  $\geq$  term2

LE: term1  $\leq$  term2

Note: For string literals, only EQ and NE conditions can be used.

Example:

; The source statements are iteratively expanded while COUNT is not zero.

```
COUNT    .ASSIGNA    2
          .AWHILE    \&COUNT NE 0    ; COUNT is set to 2.
          ADD.B      R0L,R1L          ; Condition is satisfied when COUNT is not zero.
          ADD.B      R0L,R2L
          INC.B       R0L
COUNT   .ASSIGNA    \&COUNT-1      ; COUNT minus 1.
          .AENDW
```

; The source statements are iteratively expanded while STOP is 10 or less.

```
STOP     .ASSIGNA    0
          .AWHILE    \&STOP LE 10     ; 0 is set to STOP.
          ADD.B      R0L,R1L          ; Condition is satisfied when STOP is 10 or less.
          ADD.B      R0L,R2L
          INC.B       R0L
STOP     .ASSIGNA    \&STOP+3         ; 3 is added to STOP.
          .AENDW
```

## .EXITM

Description Format: Δ.EXITM

The label field is not used.

**Description:** .EXITM terminates an iterated expansion (.AREPEAT to .AENDR) or a conditional iterated expansion (.AWHILE to .AENDW).  
Each expansion is terminated when this directive appears.  
This directive is also used to exit from macro expansions. The location of this directive must be specified carefully when macro instructions and iterated expansion are combined.

**Example:**

```

COUNT .ASSIGNA 0           ; 0 is set to COUNT.
      .AWHILE 1 EQ 1        ; An infinite loop (condition is always satisfied) is
      ADD.W   R0,R1          ; specified.
      ADD.W   R2,R3
COUNT .ASSIGNA \&COUNT+1  ; 1 is added to COUNT.
      .AIF    \&COUNT EQ 2 ; Condition: COUNT = 2
      .EXITM                ; When the condition is satisfied
      .AENDI                ; .AWHILE expansion is terminated.
      .AENDW

```

When COUNT is updated and satisfies the condition specified with the .AIF directive, .EXITM is assembled. When .EXITM is assembled, .AWHILE expansion is terminated.

The expansion results are as follows:

```

ADD.W  R0,R1 ..... When COUNT is 0
ADD.W  R2,R3
ADD.W  R0,R1 ..... When COUNT is 1
ADD.W  R2,R3

```

After this, COUNT becomes 2 and expansion is terminated.

## **.AERROR**

Description Format: Δ.AERROR

The label field is not used.

Description: When .AERROR is assembled, it generates error 670 and terminates the assembler abnormally.

This directive is also used to check the value of the preprocessor variable.

Example:

~

```
.AIF \&FLAG EQ 1
```

```
ADD.W      R0,R1
```

```
INC.W      R0
```

```
.AELSE
```

```
.AERROR           ; When \&FLAG is not 1, an error occurs.
```

```
.AENDI
```

~



## .ALIMIT

Description Format:  $\Delta$ .ALIMIT $\Delta$ <count>

The label field is not used.

Description: .ALIMIT determines the maximum count for the conditional iterated expansion (.AWHILE to .AENDW).

<count> must be specified in the following format:

- Constant (integer constant, character constant)
- Defined preprocessor variable
- Expression in which a constant or a defined preprocessor variable is used as the term

During conditional iterated (.AWHILE to .AENDW) expansion, if the statement expansion count exceeds the maximum value specified by the .ALIMIT directive, warning 854 is generated and the expansion is terminated.

If the .ALIMIT directive is not specified, the maximum count is 65,535. The maximum count of iteration expansion can be changed by respecifying this directive. The respecification is valid for the source statements after this directive.

Example:

```
COUNT .ASSIGNA 3 ; 3 is set to COUNT.
      .ALIMIT 10 ; 10 is specified as the maximum count.
      .AWHILE \&COUNT NE 4
      ADD.W R0,R1 ; [1]
      ADD.W R0,R1 ; [1]
      INC.W R0 ; [1]
COUNT .ASSIGNA \&COUNT-1 ; [1]
      .AENDW
```

[1] is expanded while COUNT is not 4. After expanding 10 times, the warning 854 is output, and the iterative expansion is terminated.

## 11.6 Macro Function

### 11.6.1 Overview of the Macro Function

The macro function allows commonly used sequences of instructions to be named and defined as one macro instruction. This is called a macro definition. Macro instructions are defined as follows:

```
~  
.MACRO <macro name>  
    <macro body>  
.ENDM  
~
```

A macro name is the name assigned to a macro instruction, and a macro body is the statements to be expanded as the macro instruction.

Using a defined macro instruction by specifying the name is called a macro call. Macro call is as follows:

```
~  
<defined macro name>  
~
```

An example of macro definition and macro call is shown below.

Example:

```
~  
.MACRO    SUM                                ; Processing to obtain the sum of R1, R2,  
ADD.W    R2,R1                            ; and R3 is defined as macro instruction SUM.  
ADD.W    R3,R1  
.ENDM  
~  
  
SUM                                ; This statement calls macro instruction SUM.  
                                ; Macro body  ADD.W  R2,R1  
                                ;              ADD.W  R3,R1  
                                ; is expanded from the macro instruction.
```

Parts of the macro body can be modified when expanded by the following procedure:

(1) Macro definition

Define arguments after the macro name in the .MACRO directive.

Use the arguments in the macro body. Arguments must be identified in the macro body by placing a backslash (\) in front of them.

(2) Macro call

Specify macro parameters in the macro call.

When the macro instruction is expanded, the arguments are replaced with their corresponding macro parameters.

Example:

```

~
.MACRO  SUM  ARG1                ; Argument ARG1 is defined.
MOV.W  R1 , \ARG1                ; ARG1 is referenced in the macro body.
ADD.W  R2 , \ARG1
ADD.W  R3 , \ARG1
.ENDM

~

SUM  R0                          ; This statement calls macro instruction SUM
                                ; specifying macro parameter R0.
                                ; The argument in the macro body is
                                ; replaced with the macro parameter, and
                                ;      ADD.W  R1,R0
                                ;      ADD.W  R2,R0
                                ;      ADD.W  R3,R0  is expanded.
```

## 11.6.2 Macro Function Directives

This assembler provides the following macro function directives.

**Table 11.16 Macro Function Directives**

Directive	Description
.MACRO	Defines a macro instruction.
.ENDM	
.EXITM	Terminates macro instruction expansion. Refer to section 11.5.2, .EXITM.

## .MACRO, .ENDM

Description Format:  $\Delta$ .MACRO $\Delta$ <macro name>[ $\Delta$ <argument>[,...]]  
 $\Delta$ .ENDM

<argument>: <argument>[=<default argument>]

The label field is not used.

Description: .MACRO and .ENDM define a macro instruction (a sequence of source statements that are collectively named and handled together).  
Naming as a macro instruction the source statements (macro body) between the .MACRO and .ENDM directives is called a macro definition.  
The operand must be specified as follows:  
.MACRO: Macro instruction, argument, or default (can be omitted)  
.ENDM: Operand field cannot be used.

(1) Macro name  
Macro names are the names assigned to macro instructions.  
Arguments are specified so that parts of the macro body can be replaced by specific parameters at expansion. Arguments are replaced with the string literals (macro parameters) specified at macro expansion (macro call).  
In the macro body, arguments are specified for replacement. The syntax of argument is macro body is as follows:  
`\<argument name>[ ' ]`  
To clearly distinguish the argument name from the rest of the source statement, an apostrophe (') can be added.

(2) Argument  
Defaults for arguments can be specified in macro definitions. The default specifies the string literal to replace the argument when the corresponding macro parameter is omitted in a macro call.  
The syntax of the argument is the same as that of symbol. The maximum length of the argument is 32 characters, and uppercase and lowercase letters are distinguished.

### (3) Default argument

The default must be enclosed with double quotation marks (") or angle brackets (<>) if any of the following characters are included in the default.

- Space
- Tab
- Comma (,)
- Semicolon (;)
- Double quotation marks (")
- Angle brackets (<>)

The assembler inserts defaults at macro expansion by removing the double quotation marks or angle brackets that enclose the string literals.

### (4) Restrictions

Macros cannot be defined in the following locations:

- Macro bodies (between .MACRO and .ENDM directives)
- Between .AREPEAT and .AENDR directives
- Between .AWHILE and .AENDW directives

The .END directive cannot be used within a macro body.

No symbol can be inserted in the label field of the .ENDM directive.

The .ENDM directive is ignored if a symbol is written in the label field, but no error is generated in this case.

Example:

; Processing to obtain the sum of R3, R4, R5 is defined as macro instruction SUM.

~

```
.MACRO  SUM
MOV.W  R3,R1
ADD.W  R4,R1
ADD.W  R5,R1
.ENDM
```

~

```
SUM                                ; This statement calls macro instruction SUM
                                   ; Macro body  MOV.W  R3,R1
                                   ;             ADD.W  R4,R1
                                   ;             ADD.W  R5,R1 is expanded.
```

; Processing to output the sum of arguments P1, P2, and P3 is defined as macro instruction TOTAL.

~

```
.MACRO  TOTAL  P1,P2,P3
MOV.W  \P1,R0
ADD.W  \P2,R0
ADD.W  \P3,R0
.ENDM
```

~

```
TOTAL  R1,R2,R3                   ; This statement calls macro instruction TOTAL.
                                   ; Macro body  MOV.W  R1,R0
                                   ;             ADD.W  R2,R0
                                   ;             ADD.W  R3,R0 is expanded.
```

### 11.6.3 Macro Body

The source statements between the .MACRO and .ENDM directives are called a macro body. The macro body is expanded and assembled by a macro call.

#### (1) Argument reference

Arguments are used to specify the parts to be replaced with macro parameters at macro expansion.

The syntax of argument reference in macro bodies is as follows:

\<argument name>[ ' ' ]

To clearly distinguish the argument name from the rest of the source statement, add an apostrophe (').

Example:

```
.MACRO  PLUS1  P,P1          ; P and P1 are arguments.
ADD     #1,\P1              ; Argument P1 is referenced.
.SDATA  "\P'1"              ; Argument P is referenced.
.ENDM
PLUS1   R,R1                ; PLUS1 is expanded.
~
```

Expanded results are as follows:

```
ADD.W   #1,R1              ; Argument P1 is referenced.
.SDATA  "R1"               ; Argument P is referenced.
```



## (2) Preprocessor variable reference (.ASSIGNA, .ASSIGNC)

Preprocessor variables can be referenced in a macro body.

The syntax for preprocessor variable reference is as follows:

`&<preprocessor variable name>[' ']`

To clearly distinguish the preprocessor variable name from the rest of the source statement, add an apostrophe (').

Example:

```
.MACRO  PLUS1
ADD      #1,R\&V1          ; Preprocessor variable V1 is referenced.
.SDATA   "\&V'1"           ; Preprocessor variable V is referenced.
.ENDM

V:        .ASSIGNC  "R"      ; Preprocessor variable V is defined.
V1:       .ASSIGNA  1        ; Preprocessor variable V1 is defined.
PLUS1     ; PLUS1 is expanded.
```

Expanded results are as follows:

```
ADD      #1,R1              ; Preprocessor variable V1 is referenced.
.SDATA   "R1"               ; Preprocessor variable V is referenced.
```

## (3) Macro generation number

The macro generation number facility is used to avoid the problem that symbols used within a macro body will be multiply defined if the macro is expanded multiple times. To avoid this problem, specify the macro generation number marker as part of any symbol used in a macro. This will result in symbols that are unique to each macro call.

The macro generation number marker is expanded as a 5-digit decimal number (between 00000 and 99999) unique to the macro expansion.

The syntax for specifying the macro generation number marker is as follows:

`\@`

Two or more macro generation number markers can be written in a macro body, and they will be expanded to the same number in one macro call.

Because macro generation number markers are expanded to numbers, they must not be written at the beginning of symbol names.

Example:

```
.MACRO    MCO,Rn
MOV.W    \Rn,\Rn
BEQ      LAB\@:8
MOV.W    #H'0,\Rn
LAB\@:    INC.W    \Rn
.ENDM
```

```
        MCO      R1
```

```
;
```

; Different symbols are created each time MCO is  
; expanded.

```
        MCO      R2
```

Expanded results are as follows:

```
        MOV.W    R1,R1
        BEQ      LQB00000:8
        MOV.W    #H'0,R1
LAB00000:
        INC.W    R1
;
        MOV.W    R2,R2
        BEQ      LQB00001:8
        MOV.W    #H'0,R2
LAB00001:
        INC.W    R2
```

#### (4) Macro replacement processing exclusion

When a backslash (\) appears in a macro body, it specifies macro replacement processing. Therefore, a means for excluding this macro processing is required when it is necessary to use the backslash as an ASCII character.

The syntax for macro replacement processing exclusion is as follows:

\(<macro replacement processing excluded string literal>)

The backslash and the parentheses will be removed in macro processing.

Example:

```
.MACRO  BACK_SLASH_SET
\ (MOV.W    #"\",R0)      ; \ is expanded as an ASCII character.
.ENDM

BACK_SLASH_SET
```

Expanded results are as follows:

```
MOV.W    #"\",R0      ; \ is expanded as an ASCII character.
```

#### (5) Comment in macro

Comments in macro bodies can be coded as normal comments or as macro internal comments. When comments in the macro body are not required in the macro expansion code, those comments can be coded as macro internal comments to suppress their expansion.

The syntax for macro internal comments is as follows:

\;<comment>

Example:

```
.MACRO  COMMENT_IGNORE Rn
MOV.W    \Rn,@-SP      \; Saves the \Rn data
.ENDM

COMMENT  IGNORE_R1
```

Expanded results are as follows (the comment is not expanded):

```
MOV.W    R1,@-SP
```

## (6) String literal manipulation functions

String literal manipulation functions can be used in a macro body. The following string literal manipulation functions are provided.

.LEN	String literal length.
.INSTR	String literal search.
.SUBSTR	String literal extraction.

### 11.6.4 Macro Call

Expanding a defined macro instruction is called a macro call. The syntax for macro calls is as follows:

Description Format:

```
[<symbol>[:]]•Δ<macro name>[Δ<macro parameter> [,...]]  
<macro parameter>: [=<argument name>]=<string literal>
```

The macro name must be defined (.MACRO) before a macro call. String literals must be specified as macro parameters to replace arguments at macro expansion. The arguments must be declared in the macro definition with .MACRO.

Description:

#### 1. Macro parameter specification

Macro parameters can be specified by either positional specification or keyword specification.

#### 2. Positional specification

The macro parameters are specified in the same order as that of the arguments declared in the macro definition with .MACRO.

#### 3. Keyword specification

Each macro parameter is specified following its corresponding argument, separated by an equal sign (=).

#### 4. Macro parameter syntax

Macro parameters must be enclosed with double quotation marks (") or angle brackets (<>) if any of the following characters are included in the macro parameters:

- Space
- Tab
- Comma (,)
- Semicolon (;)
- Double quotation marks (")
- Angle brackets (< >)

Macro parameters are inserted by removing the double quotation marks or angle brackets that enclose string literals at macro expansion.

Example:

```

        .MACRO SUM    FROM=0, TO=6          ; Macro instruction SUM and arguments
                                           ; FROM and TO are defined.

COUNT  MOV.W        R\FROM,R0
        .ASSIGNA     \FROM+1
        .AWHILE      \&COUNT LE \TO
COUNT  AND.W        R\&COUNT,R0          ; Macro body is coded using arguments
        .ASSIGNA     \&COUNT+1
        .AENDW
        .ENDW

SUM      0,3                      ; Both will be expanded into same statements.
SUM      TO=3                      ;

```

Expanded results are as follows (the arguments in the macro body are replaced with macro parameters):

```

MOV.W    R0,R0
AND.W    R1,R0
AND.W    R2,R0
AND.W    R3,R0

```

### 11.6.5 String Literal Manipulation Functions

This assembler provides the string literal manipulation functions listed in table 11.17.

**Table 11.17 String Literal Manipulation Functions**

<b>Function</b>	<b>Description</b>
.LEN	Counts the length of a string literal.
.INSTR	Searches for a string literal.
.SUBSTR	Extracts a string literal.

## .LEN

Description Format: .LEN[Δ]("<string literal>")

Description: .LEN counts the number of characters in a string literal and replaces itself with the number of characters in decimal with no radix.

When specifying a string literal, enclose the character with double quotation marks ("). When a double quotation mark is used as a character, specify two double quotation marks.

Macro arguments and preprocessor variables can be specified in the string literal as shown below.

```
.LEN( "\<argument>" )  
.LEN( "&<preprocessor variable>" )
```

This function can only be used within a macro body (between .MACRO and .ENDM directives).

Example:

```
~  
.MACRO RESERVE_LENGTH P1  
.SRES    .LEN( "\P1" )  
.ENDM
```

```
RESERVE_LENGTH ABCDEF  
RESERVE_LENGTH ABC
```

Expanded results are as follows:

```
.SRES    6                ; "ABCDEF" has six characters.  
.SRES    3                ; "ABC" has three characters.
```

## .INSTR

Description Format: `.INSTR[Δ]("<string literal 1>","<string literal 2>"  
[,<start position>])`

Description: .INSTR searches string literal 1 for string literal 2, and replaces itself with the numerical value of the position (the top of character's position of string is 0) of the found in decimal with no radix. .INSTR is replaced with -1 if string literal 2 does not appear in string literal 1.

When specifying a string literal, enclose the character with double quotation marks ("). When a double quotation mark is used as a character, specify two double quotation marks.

The <start position> parameter specifies the search start position as a numerical value, with 0 indicating the start of string literal 1. Zero is used as default when this parameter is omitted.

Macro arguments and preprocessor variables can be specified in the string literals and as the start position as shown below.

```
.INSTR( "\<argument>", ... )
```

```
.INSTR( "\&<preprocessor variable>", ... )
```

This function can only be used within a macro body (between the .MACRO and .ENDM directives).

Example:

```
.MACRO FIND_STR P1  
.DATA.W .INSTR( "ABCDEFGH", "\P1", 0 )  
.ENDM
```

```
FIND_STR CDE
```

```
FIND_STR H
```

Expanded results are as follows:

```
.DATA.W 2 ; The start position of "CDE" is 2 (0 indicating the  
; beginning of the string) in "ABCDEFGH"  
.DATA.W -1 ; "ABCDEFGH" includes no "H".
```



## **.SUBSTR**

Description Format: `.SUBSTR[Δ]("<string literal>",<start position>,<extraction length>)`

Description: `.SUBSTR` extracts from the specified string literal a substring starting at the specified start position of the specified length. `.SUBSTR` is replaced with the extracted string literal enclosed with double quotation marks ("").

When specifying a string literal, enclose the character with double quotation marks (""). When a double quotation mark is used as a character, specify two double quotation marks.

The value of the extraction start position must be 0 or greater. The value of the extraction length must be 1 or greater.

If illegal or inappropriate values are specified for the <start position> or <extraction length> parameters, this function is replaced with a space (" ").

Macro arguments and preprocessor variables can be specified in the string literal, and as the start position and extraction length parameters as shown below.

```
.SUBSTR( "\<argument>", ... )
```

```
.SUBSTR( "&<preprocessor variable>", ... )
```

This function can only be used within a macro body (between the `.MACRO` and `.ENDM` directives).

Example:

```
.MACRO RESERVE_STR P1=0,P2
.SDATA .SUBSTR( "ABCDEFGH",\P1,\P2)
.ENDM
```

```
RESERVE_STR 2,2
```

```
RESERVE_STR ,3 ; Macro parameter P1 is omitted.
```

Expanded results are as follows:

```
.SDATA "CD"
.SDATA "ABC"
```

## 11.7 Overview of Structured Assembly

The structured assembly functions provided by this assembler expand instructions which perform testing and iteration.

Table 11.18 lists the conditions for the condition codes that are used for the structured assembly directives.

**Table 11.18 Condition Codes**

Item	Condition Codes	Comparison Type	Condition Code Specification Type
1	<EQ>	<term 1> = <term 2>	Z=1
2	<NE>	<term 1> $\neq$ <term 2>	Z=0
3	<GT>	<term 1> > <term 2> (signed comparison)	Zv(N $\oplus$ V)=0
4	<LT>	<term 1> < <term 2> (signed comparison)	N $\oplus$ V=1
5	<GE>	<term 1> $\geq$ <term 2> (signed comparison)	N $\oplus$ V=0
6	<LE>	<term 1> $\leq$ <term 2> (signed comparison)	Zv(N $\oplus$ V)=1
7	<HI>	<term 1> > <term 2> (unsigned comparison)	CvZ=0
8	<LO> <CS>	<term 1> < <term 2> (unsigned comparison)	C=1
9	<HS> <CC>	<term 1> $\geq$ <term 2> (unsigned comparison)	C=0
10	<LS>	<term 1> $\leq$ <term 2> (unsigned comparison)	CvZ=1
11	<VC>		V=0
12	<VS>		V=1
13	<PL>		N=0
14	<MI>		N=1
15	<T>		Always true
16	<F>		Always false

Notes: N ... The CCR (condition code register) N (negative) flag  
 Z .... The CCR Z (zero) flag  
 V .... The CCR V (overflow) flag  
 C ... The CCR C (carry) flag  
 v .... Logical or  
 $\oplus$  ... Logical exclusive or

## 11.7.1 Notes on Structured Assembly

The structured assembly function expands the structured assembly directives into predetermined instructions and symbols, and performs no optimizations whatsoever. Thus the values that can be specified as parameters to these directives are limited by the specifications of the instructions that are generated. Furthermore, there are cases where inefficient code and/or unnecessary symbols are generated.

### 1. Instruction Expansion

The forms of structured assembly directives that involve testing condition codes may be restricted by the statement that results from expansion of the directive.

Example:

```
.IF B (R0L<LT>#10) ; Expanded instruction will cause an error
    MOV.W R1,R2
.ENDI
```

The .IF directive is expanded to CMP instruction.

However, this .IF directive results in CMP R0L,#10, and this causes an error. To avoid this, the program must be written in the following way:

```
.IF B (#10<LT>R0L) ; Expanded to CMP #10,R0L
    MOV.W R1,R2
.ENDI
```

### 2. Symbol Expansion

Structured assembly statements generate symbols in the forms shown below.

```
.IF          _$I00000 to _$I99999
.SWITCH      _$S00000 to _$S99999
.FOR[U]      _$F00000 to _$F99999
.WHILE       _$W00000 to _$W99999
.REPEAT      _$R00000 to _$R99999
```

Accordingly, such symbols are not available to the user.

## 11.7.2 Structured Assembly Directives

Table 11.19 lists the directives for structured assembly.

**Table 11.19 List of Structured Assembly Directives**

.IF	Selective processing:
.SWITCH	The instruction is selected and executed or is passed over according to the result of a test or tests.
.FOR	Iteration of processes:
.WHILE	Iteratively executes the processes while a condition is satisfied.
.REPEAT	
.BREAK	Suspends iterative processing; processing is terminated.
.CONTINUE	Suspends iterative processing; processing continues.

## .IF

Description Format :

```
Δ.IF[.<size>][:<branch size>]Δ<condition>
[Δ.ELSE[:<branch size>]]
Δ.ENDI

<size>:      {B | W | L}
<branch size>: {8 | 16}
<condition>: {term 1 <CC> term 2 | <CC>}
<CC>:       {EQ | NE | GT | LT | GE | LE | HI | LO | HS | LS | CC | CS | VC |
              VS | PL | MI | T | F}
```

The label field is not used.

**Description :** Source statements are selected and executed based on the result of testing the condition specified in the .IF directive.

When the condition is satisfied, the source statements between the .IF and the .ELSE directives are executed, and when the condition fails, the source statements between the .ELSE and the .ENDI directives are executed.

The .ELSE directive may be omitted. When omitted, the source statements between the .IF and the .ENDI directives are executed if the condition is satisfied.

### (1) Size

The size specifiers are interpreted as follows:

**B:** Byte (1 byte)

**W:** Word (2 bytes)

**L:** Longword (4 bytes)

Byte is taken as the default when the size specifier is omitted.

### (2) Branch Size

The branch size can be specified on both the .IF and the .ELSE directives. The .IF branch size specifies the branch size from the .IF directive to the .ELSE or .ENDI directive.

The .ELSE branch size specifies the branch size from the .ELSE directive to the .ENDI directive.

The following branch sizes can be specified.

Operation	Branch Size
8	8 bits
16	16 bits

Refer to section 11.3, `.DISPSIZE FBR`, and section 3.3.2, `br_relative`, and section 3.3.2, `[no]optimize`, for the setting used when the branch size specification is omitted.

Refer to table 11.18, Condition Codes, for details on the condition code conditions.

There are two types of conditions as follows:

1. Comparison type

In the comparison type, a decision is made based on a condition code based comparison of two terms.

The terms must have addressing modes that can be used with the `CMP` instruction.

2. Condition code specification type

In the condition code specification type, a decision is made based on the specified CCR (condition code register) state.

Limitations:

1. “L” cannot be specified as the size with the H8/300 and H8/300L microcomputers.
2. The value 16 cannot be specified as the branch size with the H8/300 and H8/300L microcomputers.
3. The size of the code generated by the source statements between an `.IF` directive and an `.ELSE` directive, between an `.ELSE` directive and an `.ENDI` directive, or between an `.IF` directive and an `.ENDI` directive (when the `.ELSE` directive is omitted) cannot exceed the range corresponding to the specified branch size.

The maximum source code size for the different branch sizes are as follows:

8: About 100 bytes

16: About 32,700 bytes

4. When this directive is used, symbols from `_$I00000` to `_$I99999` may be generated. Thus these symbols should not be used in programs which use the `.IF` directive.

Examples : 1. .IF.W (R0L<EQ>R1)

```
        ADD.B    #1,R0    ; [1]
        MOV.W    R0,R2    ; [1]

    .ELSE

        ADD.W    #1,R1    ; [2]
        MOV.W    R1,R2    ; [2]

    .ENDI
```

This is an example of the comparison type condition.

When R0 is equal to R1, statements [1] will be executed, and when R0 is not equal to R1, statements [2] will be executed.

2. .IF.B (#H'10<LT>R0L)

```
        SUB.W    R1,R1    ; [3]
        MOV.W    R1,R2    ; [3]

    .ENDI
```

This is an example of the comparison type condition.

Statements [3] will be executed when H'10 is less than R0L (under a signed comparison).

3. .IF (<NE>)

```
        ADD.B    #5:8,R0L ; [4]

    . ELSE

        MOV.B    R0L,R1L ; [5]

    . ENDI
```

This is an example of the condition code specification type condition.

When the CCR (condition code register) Z (zero) flag is 0, statement [4] will be executed, and when 1, statement [5] will be executed.

```

4.  .IF.B (#0<LE>R0L)
      .IF.B (#50<GE>R0L)
          MOV.W    R2,R1    ; [6]
          MOV.W    R3,R1    ; [6]
      .ENDI
  .ENDI

```

This is an example of a nested .IF construction.

If the condition  $0 \leq R0L \leq 50$  is satisfied under signed comparison, then statements [6] will be executed.



## .SWITCH

Description Format : `Δ.SWITCH[.<size>]Δ<condition1>`  
`(Δ.CASE[:<branch size>]Δ <condition2>`  
`[Δ.BREAK[:<branch size>]Δ)][...]`  
`[Δ.OTHERS]`  
`Δ.ENDS`  
`<Size>: {B | W | L}`  
`<branch size>: {8 | 16}`  
`<condition1>: {<register> | <CC>}`  
`<condition2>: {<term> | <CC> }`  
`<CC>: {EQ | NE | GT | LT | GE | LE | HI | LO | HS | LS | CC | CS | VC | VS`  
`| PL | MI | T | F}`

The label field is not used.

Description : Source statements are selected and executed based on the result of testing the conditions specified in the .SWITCH and .CASE directives.

When the condition specified by a .SWITCH directive and a corresponding .CASE directive are satisfied, the source statements between that .CASE directive and its corresponding .BREAK directive are executed.

The .SWITCH and .CASE conditions are tested in order.

When a .BREAK directive is omitted, execution continues to the statements between the next .CASE and .BREAK, or to the following statements between .OTHERS and .ENDS.

### (1) Size

The size specifies the size of the registers and terms compared in a comparison type condition. When operation size is omitted, `.SWITCH.B` (byte size) is taken as the default. It has no meaning with condition code specification type conditions.

The size specifiers are interpreted as follows:

B: Byte (1 byte)

W: Word (2 bytes)

L: Longword (4 bytes)

### (2) Branch size

The branch size can be specified on both the `.CASE` and the `.BREAK` directives.

The `.CASE` branch size specifies the branch size from the `.CASE` directive to the next `.CASE`, `.OTHERS`, or `.ENDS` directive.

The `.BREAK` branch size specifies the branch size from the `.BREAK` directive to the `.ENDS` directive.

The following branch sizes can be specified.

Operation	Branch Size
8	8 bits
16	16 bits

Refer to section 11.3, `.DISPSIZE FBR`, section 3.2.2, `br_relative`, and section 3.2.2, `[no]optimize`, for the setting used when the branch size specifier is omitted.

Refer to table 11.18, Condition Codes, for details on the condition code conditions.

There are two types of conditions as follows:

1. Comparison type

In the comparison type, a register and a term are tested for equality.

The register is specified in the `.SWITCH` directive.

The term is specified in the `.CASE` directive using an addressing mode that can be used as the source operand in the `CMP` instruction.

2. Condition code specification type

In the condition code specification type, a decision is made based on the specified CCR (condition code register) state.

CCR is specified in the `.SWITCH` directive.

The condition code(s) are specified in the `.CASE` directive(s).

Limitations:

1. “L” cannot be specified as the size with the H8/300 and H8/300L microcomputers.
2. The value 16 cannot be specified as the branch size with the H8/300 and H8/300L microcomputers.
3. The size of the code generated by the source directives corresponding to each `.CASE` directive and the size of the code between a `.BREAK` directive and the corresponding `.ENDS` directive cannot exceed the range corresponding to the specified branch size.

The maximum source code size for the different branch sizes are as follows:

8: About 100 bytes

16: About 32,700 bytes

4. When this directive is used, symbols from `_$S00000` to `_$S99999` may be generated. Thus these symbols should not be used in programs which use the `.SWITCH` directive.

Examples : 1.

```
.SWITCH.B (R0L)
.CASE #0
    MOV.W R1,R4    ; [1]
.BREAK
.CASE #1
    MOV.W R2,R4    ; [2]
.BREAK
.OTHERS
    MOV.W R3,R4    ; [3]
.ENDS
```

This is an example of the comparison type condition.

When R0L is equal to 0, statement [1] will be executed, and when R0L is equal to 1, statement [2] will be executed, and in all other cases, statement [3] will be executed.

2.

```
.SWITCH (CCR)
.CASE <CS>
    MOV.W R0,R3    ; [4]
.BREAK
.CASE <MI>
    MOV.W R1,R3    ; [5]
.ENDS
```

This is an example of the condition code type condition.

When the CCR (condition code register) C (carry) flag is 1, statement [4] will be executed, and when the N (negative) flag is 1, statement [5] will be executed.

```
3.      .SWITCH.B (R0L)
        .CASE #0
        .CASE #1
        .CASE #2
                MOV.W  R1,R3      ; [6]
        .BREAK
        .CASE #3
                MOV.W  R2,R3      ; [7]
        .ENDS
```

This is an example of omitting the .BREAK for the .CASE .

When R0L is equal to 0, 1, or 2, statement [6] will be executed, and when R0L is 3, statement [7] will be executed.

## .FOR[U]

Description Format :      Δ.FOR[U][.<size>][:<branch size>]Δ<condition>

Δ.ENDF

<size>: {B | W | L}

<branch size>: {8 | 16}

<condition>: (<loop counter>=<initial value>,<end value>[, [{ + | - }]<increment value>])

The label field is not used.

Description :      The condition specified by the loop counter and end value is tested, and the source statements between the .FOR[U] and .ENDF directives are iterated while that condition is satisfied.

There are two forms of the .FOR[U] directive: the .FOR directive, which iterates using a signed range test, and the .FORU directive, which iterates using an unsigned range test.

### (1) Size

The size specification specifies the size of the loop counter, initial value, end value and increment value.

The size specifiers are interpreted as follows:

B: Byte (1 byte)

W: Word (2 bytes)

L: Longword (4 bytes)

### (2) Branch Size

Byte is taken as the default when the size specifier is omitted.

The branch size specifies the branch size from the .FOR[U] directive to the .ENDF directive.

The following branch sizes can be specified.

Operation	Branch Size
8	8 bits
16	16 bits

Refer to section 11.3, .DISPSIZE FBR, section 3.2.2, br\_relative, and section 3.2.2, [no]optimize, for the setting used when the branch size specifier is omitted.

The operands are interpreted as follows:

- (1) <loop counter>=<initial value>

This specifies the loop counter's initial value.

The loop counter must be a register.

The initial value must have an addressing mode that can be specified as the source operand of the MOV instruction.

- (2) <end value>

The end value is the value which is compared with the loop counter.

There are two types of iteration conditions as follows:

Positive increment direction: <loop counter>  $\leq$  <end value>

Negative increment direction: <loop counter>  $\geq$  <end value>

The end value must have an addressing mode that can be specified as the source operand of the CMP instruction.

- (3) <increment value>

The increment value is the amount the loop counter is incremented or decremented on each loop iteration.

The increment direction is specified by a plus (+) to indicate a positive increment direction and a minus (–) to indicate a negative decrement direction.

Plus (+) is taken as the default when no increment direction is specified.

The increment value must have an addressing mode that can be specified as the source operand for the ADD and SUB instructions.

The value +#1 is used as the default when no increment value is specified.

The following table indicates the possible ranges of the loop counter value. Pay careful attention to the loop counter range, since infinite loops can result from inappropriate values.

Directive	Increment Direction	Size	Loop Counter Range (Initial Value to End Value)
.FOR	+	B	–128 to 126
		W	–32,768 to 32,766
		L	–2,147,483,647 to 2,147,483,646
	–	B	127 to –127
		W	32,767 to –32,767
		L	2,147,483,647 to –2,147,483,647
.FORU	+	B	0 to 254
		W	0 to 65,534
		L	0 to 4,294,967,294
	–	B	255 to 1
		W	65,535 to 1
		L	4,294,967,295 to 1

Limitations:

1. “L” cannot be specified as the size with the H8/300 and H8/300L microcomputers.
2. The value 16 cannot be specified as the branch size with the H8/300 and H8/300L microcomputers.
3. The size of the code generated by the source statements between a .FOR[U] directive and its corresponding .ENDF directive cannot exceed the range corresponding to the specified branch size.

The maximum source code size for the different branch sizes are as follows:

8: About 100 bytes

16: About 32,700 bytes

4. When this directive is used, symbols from \_\$F00000 to \_\$F99999 may be generated. Thus these symbols should not be used in programs which use the .FOR[U] directive.



Examples : 1.                   .FOR.B (R0L=#1,#10,+#1) ; [1]  
                                   ADD.B R0L,R1L  
                                   .ENDF

This is an example of a .FOR loop.

The loop counter is R0L, the initial value is #1, the end value is #10, and the increment value is +#1.

Statement [1] will be iterated while R0L is less than or equal to 10 under a signed comparison.

2.                   .FOR.W (R0=R1,R2,-R3)  
                                   ADD.B #1:8,R5L ; [2]  
                                   .ENDF

This is an example of a .FOR loop.

The loop counter is R0, the initial value is R1, the end value is R2, and the increment value is -R3.

Statement [2] will be iterated while R0 is greater than or equal to R2 under a signed comparison.

3.                   .FORU.B (R0L=#1,#200,+#1)  
                                   ADD.W R1,R2 ; [3]  
                                   ADD.W R3,R4 ; [3]  
                                   .ENDF

This is an example of a .FORU loop.

The loop counter is R0L, the initial value is #1, the end value is #200, and the increment value is +#1.

Statements [3] will be iterated while R0L is less than or equal to 200 under an unsigned comparison.

```

4.          .FORU.L (ER0=#H'00000100,#H'000001FC,+#4)

              MOV.L  @ER0,ER2                ; [4]
              MOV.L  ER2,@(H'00001100:32,ER1) ; [4]
              ADDS.L  #4,ER1                  ; [4]

          .ENDF

```

This is an example of a .FORU loop.

The loop counter is ER0, the initial value is #H'00000100, the end value is #H'000001FC, and the increment value is +#4.

Statements [4] will be iterated while ER0 is less than or equal to #H'000001FC under an unsigned comparison.

**.WHILE**

Description Format	<code>.WHILE[.size][:&lt;branch size&gt;]Δ&lt;condition&gt;</code> <code>.ENDW</code>
<size>:	{ <u>B</u>   W   L }
<branch size>:	{ 8   16 }
<condition>:	{ ( <term 1> <cc> <term 2> )   (<cc> ) }
<CC>:	{ EQ   NE   GT   LT   GE   LE   HI   LO   HS   LS   CC   CS   VC   VS   PL   MI   T   F }

Description : The condition specified in the .WHILE directive is tested, and the source statements between the .WHILE and .ENDW directives are iterated while that condition is true.

Size and branch size are as follows:

(1) Size

The size specifies the size of the terms compared in a comparison type condition. Byte is taken as the default when the size specifier is omitted. It has no meaning with condition code specification type conditions.

The size specifiers are interpreted as follows:

- B: Byte (1 byte)
- W: Word (2 bytes)
- L: Longword (4 bytes)

(2) Branch Size

The branch size specifies the branch size from the .WHILE directive to the .ENDW directive.

The following branch sizes can be specified.

Operation	Branch Size
8	8 bits
16	16 bits

Refer to section 11.3, .DISPSIZE FBR, section 3.2.2, br\_relative, and section 3.2.2, [no]optimize, for the setting used when the branch size specification is omitted.

Refer to table 11.18, Condition Codes, for details on the condition code conditions.

There are two types of conditions as follows:

1. Comparison type

In the comparison type, a decision is made based on a condition code based comparison of two terms.

The terms must have addressing modes that can be used with the CMP instruction.

2. Condition code specification type

In the condition code specification type, a decision is made based on the specified CCR (condition code register) state.

Limitations:

1. "L" cannot be specified as the size with the H8/300 and H8/300L microcomputers.
2. The value 16 cannot be specified as the branch size with the H8/300 and H8/300L microcomputers.
3. The size of the code generated by the source statements between a .WHILE directive and its corresponding .ENDW directive cannot exceed the range corresponding to the specified branch size.

The maximum source code size for the different branch sizes are as follows:

8: About 100 bytes

16: About 32,700 bytes

4. When this directive is used, symbols from \_\$W00000 to \_\$W99999 may be generated. Thus these symbols should not be used in programs which use the .WHILE directive.

Examples :1.

```
.WHILE.B (#50<GT>R0L)
    ADD.W    R1,R2        ; [1]
    ADD.B    #1:8,R0L     ; [1]
.ENDW
```

This is an example of the comparison type condition.

Statements [1] will be iterated while 50 is greater than R0L under signed comparison.

```
2.      .WHILE.W (R0<LS>R1)
        SUB.B    R2L,R3L    ; [2]
        SUB.W    R5,R1      ; [2]
.ENDW
```

This is an example of the comparison type condition.

Statements [2] will be iterated while R0 is less than or equal to R1 under unsigned comparison.

```
3.      .WHILE (<NE>)
        MOV.L    @ER2,ER4    ; [3]
        MOV.L    ER4,@ER3    ; [3]
        ADDS.L   #4,ER2      ; [3]
        ADDS.L   #4,ER3      ; [3]
        SUB.B    R1L,R0L     ; [3]
.ENDW
```

This is an example of the condition code specification type condition.

Statements [3] will be iterated while the CCR (condition code register) Z (zero) flag is 0.

```
4.      .WHILE (<PL>)
        MOV.L    ER2,@ER1    ; [4]
        ADDS.L   #4,ER1      ; [4]
        MOV.L    ER3,@ER1    ; [4]
        ADDS.L   #4,ER1      ; [4]
        ADD.W    #-1,R0      ; [4]
.ENDW
```

This is an example of the condition code specification type condition.

Statements [4] will be iterated while the CCR (condition code register) N (negative) flag is 0.

## .REPEAT

Description Format :    Δ.REPEAT

Δ.UNTIL[.<size>]Δ<condition>

<size>:       {B | W | L}

<condition>:{ (<term 1> <cc> <term 2>)| (<cc>) }

<CC>:       {EQ | NE | GT | LT | GE | LE | HI | LO | HS | LS | CC | CS |  
              VC | VS | PL | MI | T | F}

The label field is not used.

Description :       The source statements between the .REPEAT and .UNTIL directives are iterated until the condition specified in the .UNTIL directive is satisfied.

The source statements between the .REPEAT and the .UNTIL directives are executed at least once so that the .UNTIL condition can be tested.

The size specifies the size of the terms compared in a comparison type condition. .UNTIL.B (byte size) is taken as the default when the size specifier is omitted. It has no meaning with condition code specification type conditions.

The size specifiers are interpreted as follows:

B: Byte (1 byte)

W: Word (2 bytes)

L: Longword (4 bytes)

Refer to table 11.18, Condition Codes, for details on the condition code conditions.

There are two types of conditions as follows:

### 1. Comparison type

In the comparison type, a decision is made based on a condition code based comparison of two terms.

The terms must have addressing modes that can be used with the CMP instruction.

### 2. Condition code specification type

In the condition code specification type, a decision is made based on the specified CCR (condition code register) state.

- Limitations:
1. “L” cannot be specified as the size with the H8/300 and H8/300L microcomputers.
  2. The size of the code generated by the source statements between the .REPEAT and .UNTIL directives is as follows.

H8/300	: About 100 bytes
H8/300L	: About 100 bytes
Others	: About 32,700 bytes

3. When this directive is used, symbols from \_\$R00000 to \_\$R99999 may be generated. Thus these symbols should not be used in programs which use the .REPEAT directive.

- Examples :
1. .REPEAT

```
MOV.L    @ER0,ER2    ; [1]
MOV.L    ER2,@ER1    ; [1]
ADDS.L    #4,ER0      ; [1]
ADDS.L    #4,ER1      ; [1]
.UNTIL.L  (#H'001000<LS>ER0)
```

This is an example of the comparison type condition.

Statements [1] will be iterated until H'001000 is less than or equal to ER0 under unsigned comparison.

2. .REPEAT

```
ADD.W    R2,R3    ; [2]
ADD.W    R2,R4    ; [2]
SUB.B    R1L,R0L  ; [2]
.UNTIL    (<EQ>)
```

This is an example of the condition code specification type condition.

Statements [2] will be iterated until the CCR (condition code register) Z (zero) flag is 1.

## .BREAK

Description Format :     Δ.BREAK[:<branch size>]  
                          <branch size>: {8 | 16}  
                          The label field is not used.

Description :     The .BREAK directive terminates .FOR[U], .WHILE, and .REPEAT loops, exiting the loop without executing the source statements following the .BREAK directive. More specifically, the .BREAK directive executes an unconditional jump to the .ENDF, .ENDW, or .UNTIL directive that closes the corresponding .FOR[U], .WHILE, or .REPEAT loop, thus terminating the processing.

The branch size specifies the branch size from the .BREAK directive to the corresponding .ENDF, .ENDW, or .UNTIL directive.

The following branch sizes can be specified.

Operation	Branch Size
8	8 bits
16	16 bits

Refer to section 11.3, .DISPSIZE FBR, section 3.2.2, br\_relative, and section 3.2.2, [no]optimize, for the setting used when the branch size specification is omitted.

This directive can also be used with the .SWITCH directive.

Refer to section 11.7, .SWITCH, for details on use of the .SWITCH directive.

Limitations:     The value 16 cannot be specified as the branch size with the H8/300 and H8/300L microcomputers.

Example :             .WHILE (<T>)  
                               .IF.B (#10<LE>R0L)  
                               .BREAK  
                               .ENDI  
                               ADD.W   R1,R2  
                               INC.B    R0L  
                          .ENDW

The iteration will terminate when 10 is less than or equal to R0L.



## .CONTINUE

Description Format : Δ.CONTINUE[:<branch size>]

<branch size>: {8 | 16}

The label field is not used.

Description : The .CONTINUE directive restarts loop processing without executing the remaining source statements in the .FOR[U], .WHILE, and .REPEAT loops. More specifically, the .CONTINUE directive branches unconditionally to the loop test point in a .FOR[U], .WHILE, or .REPEAT loop.

The branch size specifies the branch size from the .CONTINUE directive to the corresponding .ENDF, .WHILE, or .UNTIL directive.

The following branch sizes can be specified.

Operation	Branch Size
8	8 bits
16	16 bits

Refer to section 11.3, .DISPSIZE FBR, section 3.2.2, br\_relative, and section 3.2.2, [no]optimize, for the setting used when the branch size specification is omitted.

Limitations: The value 16 cannot be specified as the branch size with the H8/300 and H8/300L microcomputers.

Example :

```
.WHILE.B (#10<GT>R0L)
    INC.B    R0L
    INC.B    R1L
    .IF.B    (#10<LT>R1L)
        .CONTINUE
    .ENDI
    ADD.W    R2,R3        ; [1]
    .ENDW
```

Statement [1] will not be executed when 10 is less than R1L.



# Section 12 Compiler Error Messages

## 12.1 Error Format and Error Levels

In this section, error messages output in the following format and the details of errors are explained.

Error number            (Error level) Error message

Error details

There are five different error levels, corresponding to different degrees of seriousness.

Error Level	Error Type	Description
(I)	Information	Processing is continued and the object program is output.
(W)	Warning	Processing is continued and the object program is output.
(E)	Error	Processing is continued but the object program is not output.
(F)	Fatal	Processing is interrupted and an error message is output simultaneously.
(-)	Internal	Processing is interrupted and an error message is output simultaneously.

## 12.2 Error Messages

### **C0002 (I) No declarator**

A declaration without a declarator exists.

### **C0003 (I) Unreachable statement**

A statement that will not be executed exists.

### **C0004 (I) Constant as condition**

A constant expression is specified as the condition for an **if** or **switch** statement.

**C0005 (I) Precision lost**

Precision may be lost when assigning via type conversion from a right hand side value to the left hand side value.

**C0006 (I) Conversion in argument**

A function parameter expression is converted into a parameter type specified in the prototype declaration.

**C0008 (I) Conversion in return**

A return statement expression is converted into a value type that should be returned from a function.

**C0010 (I) Elimination of needless expression**

A needless expression exists.

**C0011 (I) Used before set symbol "variable name"**

A local variable is used before setting its value.

**C0015 (I) No return value**

A return statement has no return value or a return statement does not exist in a function which returns a value of other than the void type.

**C0016 (I) Padding in structure**

An empty space has been created between structure members by boundary alignment.

**C0100 (I) Function "function name" not optimized**

A function which is too large cannot be optimized.

**C0101 (I) Optimizing range divided in function "function name"**

The optimizing range of "function name" is divided into some blocks.

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

Any register cannot be allocated to the variable of the register storage class.

**C0200 (I) No prototype function**

There is no prototype declaration.

**C0300 (I) #pragma interrupt has no effect**

The function specified by **#pragma interrupt** is not found.

**C0301 (I) #pragma abs8 has no effect**

The variable specified by **#pragma abs8** is not found.

**C0302 (I) #pragma abs16 has no effect**

The variable specified by **#pragma abs16** is not found.

**C0303 (I) #pragma indirect has no effect**

The function specified by **#pragma indirect** is not found.

**C0304 (I) #pragma regsave/noregsave has no effect**

The function specified by **#pragma regsave/noregsave** is not found.

**C0305 (I) #pragma inline/inline\_asm has no effect**

The function specified by **#pragma inline/inline\_asm** is not found.

**C0306 (I) #pragma global\_register has no effect**

The variable specified by **#pragma global\_register** is not found.

**C0307 (I) #pragma entry has no effect**

The declaration specified by **#pragma entry** is not found.

**C0308 (I) #pragma address has no effect**

The variable specified by **#pragma address** is not found.

**C1000 (W) Illegal pointer assignment**

A pointer is assigned to a pointer with different type.

**C1001 (W) Illegal comparison in "operator"**

The operands of the binary operator `=` or `!=` are a pointer and an integer other than 0, respectively.

**C1002 (W) Illegal pointer for "operator"**

The operands of the binary operator `=`, `!=`, `>`, `<`, `>=`, or `<=` are pointers assigned to different types.

**C1005 (W) Undefined escape sequence**

An undefined escape sequence (a backslash and the character following the backslash) is used in a character constant or string literal.

**C1007 (W) Long character constant**

A character constant consists of two characters.

**C1008 (W) Identifier too long**

An identifier consists of more than 8189 characters. The 8190th and subsequent characters are ignored.

**C1010 (W) Character constant too long**

A character constant consists of more than two characters. The third and subsequent characters are ignored.

**C1012 (W) Floating point constant overflow**

The value of a floating-point constant exceeds the limit. Assumes the internally represented value corresponding to  $+\infty$  or  $-\infty$  depending on the sign of the result.

**C1013 (W) Integer constant overflow**

The value of an unsigned long integer constant exceeds the limit. Assumes a value ignoring the overflowed upper bits.

**C1014 (W) Escape sequence overflow**

The value of an escape sequence indicating a bit pattern in a character constant or string literal exceeds 255. The low order byte is valid.

**C1015 (W) Floating point constant underflow**

The absolute value of a floating-point constant is less than the lower limit. Assumes 0.0 as the value of the constant.

**C1016 (W) Argument mismatch**

The data type assigned to a pointer specified as a formal parameter in a prototype declaration differs from the data type assigned to a pointer used as the corresponding actual parameter in a function call. Uses the internal representation of the pointer used for the function call actual parameter.

**C1017 (W) Return type mismatch**

The function return type and the type of a return statement expression are pointers but the data types assigned to these pointers are different. Uses the internal representation of the pointer specified in the return statement expression.

**C1019 (W) Illegal constant expression**

The operands of the relational operator  $<$ ,  $>$ ,  $<=$ , or  $>=$  in a constant expression are pointers to different data types. Assumes 0 as the result value.

**C1020 (W) Illegal constant expression of "--"**

The operands of the binary operator  $-$  in a constant expression are pointers to different data types. Assumes 0 as the result value.

**C1021 (W) Convert to sjis-space**

Some Japanese codes cannot be converted into the specified output codes. Converts to shift-JIS spaces.

**C1022 (W) Convert to euc-space**

Some Japanese codes cannot be converted into the specified output codes. Converts to EUC spaces.

**C1023 (W) Can not convert japanese code "character" to output type**

Some Japanese codes cannot be converted into the specified output codes. Converts to spaces.

**C1024 (W) First operand of "operator" is not lvalue**

A value other than the left value is specified for the first operand of the operator.

**C1025 (W) Out of float**

The number of digits in a floating-point constant exceeds 17. The 18th and following digits are invalid.

**C1026 (W) Address of packed member**

The address of a structure member with **pack=1** specification is referred to.

**C1200 (W) Division by floating point zero**

Division by the floating-point number 0.0 is carried out in a constant expression. Assumes the internal representation value corresponding to  $+\infty$  or  $-\infty$  depending on the sign of the operands.

**C1201 (W) Ineffective floating point operation**

Invalid floating-point operations such as  $\infty - \infty$  or  $0.0/0.0$  are carried out in a constant expression. Assumes the internal representation value corresponding to a not-a-number indicating the result of an ineffective operation.

**C1300 (W) Command parameter specified twice**

The same compiler option is specified more than once. Uses the last specified compiler option.

**C1302 (W) 'frame' or 'noframe' option ignored**

The **frame** option is specified when optimization is specified, or the **noframe** option is specified when no optimization is specified. The **'frame'** or **'noframe'** option is ignored.

**C1305 (W) 'show=object' option ignored**

The **show=object** option is specified when assembly source program output is specified. The **show=object** option is ignored.

**C1306 (W) 'speed=inline' option ignored**

The **speed=inline** option is specified when no optimization is specified. The **speed=inline** option is ignored.

**C1307 (W) Section name too long**

The length of a section name exceeds 8192 characters. Uses the first 8192 characters and ignores the rest.

#### **C1308 (W) 'speed=loop' option ignored**

The **speed=loop** option is specified when no optimization is specified. The **speed=loop** option is ignored.

#### **C1310 (W) 'goptimize' option ignored**

The **goptimize** option is specified when assembly source program output is specified. The **goptimize** option is ignored.

#### **C1311 (W) 'cmncode' option ignored**

The **cmncode** option is specified when no optimization is specified. The **cmncode** option is ignored.

#### **C1313 (W) Invalid SBR value**

A value other than zero is specified for the lower eight bits in the **sbr** option. Ignores the specification of the lower eight bits.

#### **C1314 (W) 'ecpp' option ignored**

The **ecpp** option is specified when the C++ exception processing functions are enabled. The **ecpp** option is ignored.

#### **C1315 (W) 'noregexpansion' option ignored**

The **noregexpansion** option is specified when the CPU type is H8SX or H8S (withtout **legacy=v4** option). The **noregexpansion** option is ignored.

#### **C1316 (W) 'cmncode' option ignored**

The **cmncode** option is specified when the CPU type is H8SX or H8S (withtout **legacy=v4** option). The **cmncode** option is ignored.

#### **C1318 (W) 'align=4' option ignored**

The **align=4** option is specified when the CPU type is not H8SX. The **align=4** option is ignored.

#### **C1319 (W) 'speed=intrinsic' option ignored**

The **speed=intrinsic** option is specified when the CPU type is not H8SX. The **speed=intrinsic** option is ignored.

#### **C1321 (W) 'sbr' option ignored**

The **sbr** option is specified when the CPU type is not H8SX. The **sbr** option is ignored.

#### **C1322 (W) 'volatile\_loop' option ignored**

The **volatile\_loop** option is specified when the CPU type is not H8SX or H8S (withtout **legacy=v4** option). The **volatile\_loop** option is ignored.



#### **C1323 (W) 'infinite\_loop' option ignored**

The **infinite\_loop** option is specified when the CPU type is not H8SX or H8S (without **legacy=v4** option). The **infinite\_loop** option is ignored.

#### **C1324 (W) 'ptr16' option ignored**

The **ptr16** option is specified when the CPU type is not H8SX or H8S (without **legacy=v4** option). The **ptr16** option is ignored.

#### **C1325 (W) 'del\_vacant\_loop' option ignored**

The **del\_vacant\_loop** option is specified when the CPU type is not H8SX or H8S (without **legacy=v4** option). The **del\_vacant\_loop** option is ignored.

#### **C1326 (W) 'global\_alloc' option ignored**

The **global\_alloc** option is specified when the CPU type is not H8SX or H8S (without **legacy=v4** option). The **global\_alloc** option is ignored.

#### **C1327 (W) 'struct\_alloc' option ignored**

The **struct\_alloc** option is specified when the CPU type is not H8SX or H8S (without **legacy=v4** option).. The **struct\_alloc** option is ignored.

#### **C1328 (W) 'const\_var\_propagate' option ignored**

The **const\_var\_propagate** option is specified when the CPU type is not H8SX or H8S (without **legacy=v4** option). The **const\_var\_propagate** option is ignored.

#### **C1329 (W) 'opt\_range' option ignored**

The **opt\_range** option is specified when the CPU type is not H8SX or H8S (without **legacy=v4** option). The **opt\_range** option is ignored.

#### **C1330 (W) 'max\_unroll' option ignored**

The **max\_unroll** option is specified when the CPU type is not H8SX or H8S (without **legacy=v4** option). The **max\_unroll** option is ignored.

#### **C1331 (W) Section name "S" specified**

The **S** is specified for the section name. The **S** may be identified with the section name of the stack area that is generated by the compiler.

#### **C1332 (W) 'indirect = extended' option ignored**

The **indirect=extended** option is specified when the CPU type is not H8SX. The **indirect=extended** option is ignored.

#### **C1333 (W) 'enable\_register' option ignored**

300HN, 300HA, 300, 300L, or 300Reg was specified as the CPU type or the **legacy=v4** option was specified. The **enable\_register** option is ignored.

**C1334 (W) 'legacy=v4' option ignored**

The CPU type is not H8S. The **legacy=v4** option is ignored.

**C1335 (W) 'strict\_ansi' option ignored**

300HN, 300HA, 300, 300L, or 300Reg was specified as the CPU type or the **legacy=v4** option was specified. The **strict\_ansi** option is ignored.

**C1336 (W) 'cpuexpand=v6' option ignored**

The **cpuexpand=v6** option is specified when the CPU type is not H8SX or H8S (without **legacy=v4** option). The **cpuexpand=v6** option is ignored

**C1337 (W) 'noscope' option ignored**

300HN, 300HA, 300, 300L, or 300Reg was specified as the CPU type or the **legacy=v4** option was specified. The **noscope** option is ignored.

**C1338 (W) Invalid SBR value in H8SXM**

H8SXM was specified as the CPU type, so the address specified for the SBR is outside the possible range.

**C1339 (W) 'file\_inline' option ignored**

300HN, 300HA, 300, 300L, or 300Reg was specified as the CPU type or the **legacy=v4** option was specified. The **file\_inline** option is ignored.

**C1341 (W) 'file\_inline\_path' option ignored**

300HN, 300HA, 300, 300L, or 300Reg was specified as the CPU type or the **legacy=v4** option was specified. The **file\_inline\_path** option is ignored

**C1342 (W) 'character string 1' is interpreted as 'character string 2'**

'Character string 1' was specified as an option but no option was found. In compilation, 'character string 1' was interpreted as 'character string 2'.

**C1400 (W) Function "function name" in #pragma inline is not expanded**

A function specified using the **#pragma inline** could not be expanded where the function is called. Ignores the **#pragma inline** specification.

**C1401 (W) #pragma abs16 ignored**

**#pragma abs16** is specified when the CPU/operating mode is H8SXN, H8SXM, **2600n**, **2000n**, **300hn**, or **300**. Ignores the **#pragma abs16** specification.

**C1403 (W) #pragma asm ignored**

**#pragma asm** is specified when the object format is a relocatable object program. Ignores the **#pragma asm** specification.

**C1404 (W) 'case=table' option ignored by switch**

The **switch** statement cannot be expanded to the jump table method. Expands the **switch** statement to the **if\_then** method.

**C1405 (W) Illegal #pragma syntax**

An illegal **#pragma** is specified. Ignores the **#pragma** specification.

**C1406 (W) Abs8 attribute ignored**

Ignores the **abs8** specification.

**C1407 (W) #pragma address ignored**

A **#pragma** address specification is invalid for an explicitly initialized variable.

**C1510 (W) Illegal bit width**

An illegal bit width is specified with the **CPU** option.

**C1511 (W) Illegal value in operand**

A value outside the range is specified to an operand.

**C2000 (E) Illegal preprocessor keyword**

An illegal keyword is used in a preprocessor directive.

**C2001 (E) Illegal preprocessor syntax**

There is an error in a preprocessor directive or in a macro call specification.

**C2007 (E) Invalid include file name "file name"**

The specification of the include file name is invalid.

**C2016 (E) Preprocessor constant expression too complex**

The total number of operators and operands in a constant expression specified by an **#if** or **#elif** directive exceeds 512.

**C2019 (E) File name too long**

The length of a file name exceeds 4096 characters.

**C2020 (E) System identifier "name" redefined**

The name of the defined symbol is the same as that of the intrinsic function.

**C2021 (E) System identifier "name" mismatch**

An intrinsic function not corresponding to the specified CPU/operating mode is used.

**C2100 (E) Multiple storage classes**

Two or more storage class specifiers are used in a declaration.

**C2101 (E) Address of register**

A unary-operator & is used for a variable that has a register storage class.

**C2102 (E) Illegal type combination**

An illegal combination of type specifiers is used.

**C2103 (E) Bad self reference structure**

A structure or union member has the same data type as its parent.

**C2104 (E) Illegal bit field width**

A constant expression indicating the width of a bit field is not an integer or it is negative.

**C2105 (E) Incomplete tag used in declaration**

An incomplete tag name declared with a structure or union, or an undeclared tag name is used in a typedef declaration or in the declaration of a data type not assigned to a pointer or to a function return value.

**C2106 (E) Extern variable initialized**

A compound statement specifies an initial value for an extern storage class variable.

**C2107 (E) Array of function**

An array with a function type is specified.

**C2108 (E) Function returning array**

A function with an array return value type is specified.

**C2109 (E) Illegal function declaration**

A storage class other than extern is specified in the declaration of a function variable used in a compound statement.

**C2110 (E) Illegal storage class**

The storage class in an external definition is specified as **auto** or **register**.

**C2111 (E) Function as a member**

A member of a structure or union is declared as a function.

**C2112 (E) Illegal bit field**

The type specifier for a bit field is illegal. **char**, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **enum**, **bool**, or a combination of **const** or **volatile** with one of the above types is allowed as a type specifier for a bit field.

**C2113 (E) Bit field too wide**

The width of a bit field is greater than the size (8, 16, or 32 bits) indicated by its type specifier.

**C2114 (E) Multiple variable declarations**

A variable name is declared more than once in the same scope.

**C2115 (E) Multiple tag declarations**

A structure, union, or enum tag name is declared more than once in the same scope.

**C2117 (E) Empty source program**

There are no external definitions in the source program.

**C2118 (E) Prototype mismatch "function name"**

A function type differs from the one specified in the declaration.

**C2119 (E) Not a parameter name "parameter name"**

An identifier not in the function parameter list is declared as a parameter.

**C2120 (E) Illegal parameter storage class**

A storage class other than **register** is specified in a function parameter declaration.

**C2121 (E) Illegal tag name**

The combination of a structure, union, or enum with a tag name differs from the declared combination.

**C2122 (E) Bit field width 0**

The width of a bit field specifying a member name is 0.

**C2123 (E) Undefined tag name**

An undefined tag name is specified in an enum declaration.

**C2124 (E) Illegal enum value**

A non-integral constant expression is specified as a value for an enum member.

**C2125 (E) Function returning function**

A function with a function type return value is specified.

**C2126 (E) Illegal array size**

The value that specifies the number of array elements exceeds the limit

**C2127 (E) Missing array size**

The number of elements in an array is not specified.

**C2129 (E) Illegal initializer type**

The initial value specified for a variable is not a type that can be assigned to a variable.

**C2130 (E) Initializer should be constant**

A value other than a constant expression is specified either as the initial value of a structure, union or array variable, or as the initial value of a static variable.

**C2131 (E) No type nor storage class**

Storage class or type specifiers is not given in an external data definition.

**C2132 (E) No parameter name**

A parameter is declared even though the function parameter list is empty.

**C2133 (E) Multiple parameter declarations**

Either a parameter name is declared in a macro function definition parameter list more than once, or a parameter is declared inside and outside the function declarator.

**C2134 (E) Initializer for parameter**

An initial value is specified in the declaration of a parameter.

**C2135 (E) Multiple initialization**

A variable is initialized more than once.

**C2136 (E) Type mismatch**

An extern or static storage class variable, or function is declared more than once with different data types.

**C2137 (E) NULL declaration for parameter**

An identifier is not specified in the function parameter declaration.

**C2138 (E) Too many initializers**

The number of initial values specified for a structure, union, or array is greater than the number of structure members or array elements. This error also occurs if two or more initial values are specified when the first member of a union is scalar.

**C2139 (E) No parameter type**

A type is not specified in a function parameter declaration.

**C2140 (E) Illegal bit field**

A bit field is used in a union.

**C2141 (E) Struct has no member name**

An anonymous bit field is used as the first member of a structure.

**C2142 (E) Illegal void type**

**void** is used illegally. **void** can only be used in the following three cases:

- (1) To specify a type assigned to a pointer
- (2) To specify a function return value type
- (3) To explicitly specify that a function whose prototype is declared does not have a parameter

**C2143 (E) Illegal static function**

There is a function declaration with a static storage class function that has no definition in the source program.

**C2150 (E) Multiple function qualifiers**

Multiple function qualifiers are specified.

**C2151 (E) "name" must be qualified for function type**

"name" can qualify only the function type.

**C2152 (E) Illegal attribute combination**

An illegal attribute combination is specified. The following attribute combinations are allowed.

	<b>_near8</b>	<b>_near16</b>	<b>_abs8</b>	<b>_abs16</b>	<b>_ptr16</b>	<b>_interrupt</b>	<b>_inline</b>	<b>_indirect</b>	<b>_indirect_ex</b>	<b>_regsave</b>	<b>_noregsave</b>
<b>__near8</b>	x	x	O	O	x	x	x	x	x	x	x
<b>__near16</b>	x	x	O	O	x	x	x	x	x	x	x
<b>__abs8</b>	O	O	x	x	x	x	x	x	x	x	x
<b>__abs16</b>	O	O	x	x	O	x	x	x	x	x	x
<b>__ptr16</b>	x	x	x	O	x	x	x	x	x	x	x
<b>__interrupt</b>	x	x	x	x	x	x	x	x	x	O	O
<b>__inline</b>	x	x	x	x	x	x	x	O	O	x	x
<b>__indirect</b>	x	x	x	x	x	x	O	x	x	O	O
<b>__indirect_ex</b>	x	x	x	x	x	x	O	x	x	O	O
<b>__regsave</b>	x	x	x	x	x	O	x	O	O	x	x
<b>__noregsave</b>	x	x	x	x	x	O	x	O	O	x	x

Symbols: O: Allowed, x: Not allowed

### C2153 (E) Illegal "name" specifier

There is an illegal attribute specifier.

### C2154 (E) "name" must be specified for variables

This attribute specifier can be specified only for variables.

### C2155 (E) "name" must be specified for functions

This attribute specifier can be specified only for functions.

### C2157 (E) Attribute keyword and pragma cannot be specified for one symbol

An attribute keyword and **#pragma** declaration cannot be specified simultaneously.

### C2158 (E) Attribute mismatch

Attributes are mismatched between declarations.

### C2159 (E) Multiple entry functions

Multiple entry functions are specified.

### C2160 (E) Illegal ' \_\_near8/\_near16' variable size

The size of variable where \_\_near8 or \_\_near16 is specified exceeds the available range.



**C2161 (E) Illegal '\_abs8' variable type**

A variable type specified for abs8 is illegal.

**C2162 (E) Illegal '\_global\_register' variable type**

A variable type specified for \_global\_register is illegal.

**C2163 (E) Illegal '\_interrupt' function type**

An interrupt function type is illegal.

**C2164 (E) Cannot specify "name" to local storage class**

An illegal attribute is specified.

**C2165 (E) Multiple pointer qualifiers**

More than one \_ptr16 is specified.

**C2166 (E) '\_ptr16' must be qualified for data pointer type**

\_ptr16 is specified to a type other than the data pointer type.

**C2190 (E) Multiple functions on vector "vector number"**

Multiple functions are specified for a vector number.

**C2200 (E) Index not integer**

An array index expression type is not integer type.

**C2201 (E) Cannot convert parameter "n"**

The n-th parameter of a function call cannot be converted to the type of parameter specified in the prototype declaration.

**C2202 (E) Number of parameters mismatch**

The number of parameters for a function call is not equal to the number of parameters specified in the prototype declaration.

**C2203 (E) Illegal member reference for "."**

The expression to the left-hand side of the (.) operator is not a structure or union.

**C2204 (E) Illegal member reference for "->"**

The expression to the left of the -> operator is not a pointer to a structure or union.

**C2205 (E) Undefined member name**

An undeclared member name is used to reference a structure or union.

**C2206 (E) Modifiable lvalue required for "operator"**

The expression for a prefix or suffix operator ++ or -- has a left value that cannot be assigned (a left value whose type is not array or const).

**C2207 (E) Scalar required for "!"**

The unary operator ! is used in an expression that is not scalar.

**C2208 (E) Pointer required for "\*"**

The unary operator \* is used in an expression that is not a pointer or in an expression of a pointer for void.

**C2209 (E) Arithmetic type required for "operator"**

The unary operator + or – is used in a non-arithmetic expression.

**C2210 (E) Integer required for "~"**

The unary operator ~ is used in a non-integral expression.

**C2211 (E) Illegal sizeof**

A **sizeof** operator is used for a bit field member, function, void, or array with an undefined size.

**C2212 (E) Illegal cast**

Either because array, structure, or union is specified in a cast operator, or because the operand of a cast operator is void, structure, or union, the operand cannot be converted.

**C2213 (E) Arithmetic type required for "operator"**

The binary operator \*, /, \*=, or /= is used in a non-arithmetic expression.

**C2214 (E) Integer required for "operator"**

The binary operator <<, >>, &, |, ^, %, <<=, >>=, &=, |=, ^=, or %= is used in a non-integral expression.

**C2215 (E) Illegal type for "+"**

The combination of operand types used with the binary operator + is illegal. Only the following type combinations are allowed for the binary operator +:

- (1) Two arithmetic type operands
- (2) Pointer type and integer type

**C2216 (E) Illegal type for parameter**

Type void is specified for a function call parameter type.

**C2217 (E) Illegal type for "-"**

The combination of operand types used with the binary operator – is not allowed. Only the following three combinations are allowed for the binary operator:

- (1) Two arithmetic type operands
- (2) Two pointers assigned to the same data type

(3) The first operand is pointer type and the second operand is integral type.

#### **C2218 (E) Scalar required in "?:"**

The first operand of the conditional operator ?: is not scalar type.

#### **C2219 (E) Type not compatible in "?:"**

The types of the second and third operands of the conditional operator ?: do not match with each other. Only the following six combinations are allowed for the second and third operands when using the ?: operator:

- (1) Two arithmetic type operands
- (2) Two void type operands
- (3) Two pointers to the same data type
- (4) A pointer, and either an integer constant whose value is zero or another pointer to void that is converted from an integer constant whose value is zero
- (5) A pointer and another pointer to void
- (6) Two structure or union variables with the same data type

#### **C2220 (E) Modifiable lvalue required for "operator"**

An expression whose left value cannot be assigned (a left value whose type is not array or const) is used as an operand of a left assignment operator =, \*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, or |=.

#### **C2221 (E) Illegal type for "operator"**

The operand of the postfix operator ++ or -- is a pointer to type other than scalar type, to function type or to void type.

#### **C2222 (E) Type not compatible for "="**

The operand types for the assignment operator = do not match. Only the following five combinations are allowed for the operands of the assignment operator =:

- (1) Two arithmetic type operands
- (2) Two pointers to the same data type
- (3) The left operand is a pointer, and the right operand is either an integer constant whose value is zero or another pointer to void that is converted from an integer constant whose value is zero.
- (4) A pointer and another pointer to void
- (5) Two structure or union variables with the same data type

#### **C2223 (E) Incomplete tag used in expression**

An incomplete tag name is used for a structure or union in an expression.

**C2224 (E) Illegal type for assign**

The operand types of the assignment operator += or -= are illegal.

**C2225 (E) Undeclared name "name"**

An undeclared name is used in an expression.

**C2226 (E) Scalar required for "operator"**

The binary operator && or || is used in a non-scalar expression.

**C2227 (E) Illegal type for equality**

The combination of operand types for the equality operator == or != is not allowed. Only the following three combinations of operand types are allowed for the equality operator == or !=:

- (1) Two arithmetic type operands
- (2) Two pointers to the same data type
- (3) A pointer, and either an integer constant whose value is zero or another pointer to void that is converted from an integer constant whose value is zero.

**C2228 (E) Illegal type for comparison**

The combination of operand types for the relational operator >, <, >=, or <= is not allowed. Only the following two combinations of operand types are allowed for a relational operator:

- (1) Two arithmetic type operands
- (2) Two pointers to the same data type

**C2230 (E) Illegal function call**

An expression which is not a function type or a pointer to a function type is used for a function call.

**C2231 (E) Address of bit field**

The unary operator & is used in a bit field.

**C2232 (E) Illegal type for "operator"**

The operand of the prefix operator ++ or -- is a pointer to type other than scalar type, to function type or to void type.

**C2233 (E) Illegal array reference**

An expression used as an array is type other than array type or a pointer to function type or to void.

**C2234 (E) Illegal typedef name reference**

A typedef name is used as a variable in an expression.

**C2235 (E) Illegal cast**

An attempt is made to cast a pointer to a floating-point type.

**C2237 (E) Illegal constant expression**

In an expression, a pointer constant is cast to an integer, and the result is manipulated.

**C2238 (E) Lvalue or function type required for "&"**

The unary operator & is used against the lvalue or an expression other than function type.

**C2239 (E) Illegal section name**

A section name includes a character that cannot be used.

**C2240 (E) Illegal section naming**

The section is illegally named. The same name is assigned to different sections.

**C2300 (E) Case not in switch**

A **case** label is specified outside a **switch** statement.

**C2301 (E) Default not in switch**

A **default** label is specified outside a **switch** statement.

**C2302 (E) Multiple labels**

A label name is defined more than once in a function.

**C2303 (E) Illegal continue**

A **continue** statement is specified outside a **while**, **for**, or **do** statement.

**C2304 (E) Illegal break**

A **break** statement is specified outside a **while**, **for**, **do**, or **switch** statement.

**C2305 (E) Void function returns value**

A **return** statement specifies a return value for a function with a void return type.

**C2306 (E) Case label not constant**

A **case** label expression is not an integral type constant expression.

**C2307 (E) Multiple case labels**

Two or more **case** labels with the same value are specified for one **switch** statement.

**C2308 (E) Multiple default labels**

Two or more **default** labels are specified for one **switch** statement.

**C2309 (E) No label for goto**

There is no label corresponding to the destination specified by a **goto** statement.

**C2310 (E) Scalar required "while, for, do"**

The control expression (that determines statement execution) for a **while**, **for**, or **do** statement is not a scalar.

**C2311 (E) Integer required**

The control expression (that determines statement execution) for a **switch** statement is not integral type.

**C2312 (E) Missing "("**

The control expression (that determines statement execution) does not have a left parenthesis "(" for an **if**, **while**, **for**, **do**, or **switch** statement.

**C2313 (E) Missing ";"**

A **do** statement is ended without a semicolon (;).

**C2314 (E) Scalar required "if"**

A control expression (that determines statement execution) of an **if** statement is not scalar type.

**C2316 (E) Illegal type for return value**

An expression in a **return** statement cannot be converted to the type of value expected to be returned by the function.

**C2320 (E) Illegal asm position**

The position of **#pragma asm** is illegal.

**C2330 (E) Illegal #pragma interrupt function declaration**

The interrupt function declaration is illegal.

**C2331 (E) Illegal interrupt function call**

A function with an interrupt function declaration is called or referenced in the program.

**C2332 (E) Function "function name" in #pragma interrupt already declared**

The function specified by interrupt function declaration **#pragma interrupt** has already been declared as a normal function.

**C2333 (E) Multiple interrupt for one function**

Interrupt function declaration **#pragma interrupt** has been declared more than once against the same function.

**C2334 (E) Illegal parameter in #pragma interrupt function**

The parameter type used for an interrupt function is illegal. Only void can be specified for the parameter.

**C2335 (E) Missing parameter declaration in #pragma interrupt function**

An undeclared variable or function is used in stack switching specification (sp) for interrupt function declaration **#pragma interrupt**, or interrupt function termination specification (sy).

**C2336 (E) Parameter out of range in #pragma interrupt function**

The value of parameter tn for interrupt function declaration **#pragma interrupt** exceeds 3.

**C2337 (E) Illegal #pragma interrupt function type**

The interrupt function declaration is illegal.

**C2340 (E) Illegal #pragma abs8 declaration**

The short absolute address variable declaration is illegal.

**C2341 (E) Variable "variable name" in #pragma abs8 already declared**

The variable specified by short absolute address variable declaration **#pragma abs8** has already been declared as a variable.

**C2342 (E) Illegal #pragma abs8 symbol type**

The variable specified by short absolute address variable declaration **#pragma abs8** has been declared as a type other than a variable name.

**C2345 (E) Illegal #pragma abs16 declaration**

The short absolute address variable declaration is illegal.

**C2346 (E) Variable "variable name" in #pragma abs16 already declared**

The variable specified by short absolute address variable declaration **#pragma abs16** has already been declared as a variable.

**C2347 (E) Illegal #pragma abs16 symbol type**

The variable specified by short absolute address variable declaration **#pragma abs16** has been declared as a type other than a variable name.

**C2350 (E) Illegal section name declaration**

The **#pragma section** specification is illegal.

**C2352 (E) Section name table overflow**

The total number of sections exceeds 65280.

**C2353 (E) Section size overflow regarding "section name"**

The section size exceeds 32 kbytes.

**C2360 (E) Illegal #pragma indirect function declaration**

Indirect memory function declaration is illegal.

**C2361 (E) Function "function name" in #pragma indirect function already declared**

The function specified by indirect memory function declaration **#pragma indirect** has already been declared as a function.

**C2362 (E) Illegal #pragma indirect function type**

The function specified by indirect memory function declaration **#pragma indirect** has been declared or defined as a type other than a function.

**C2363 (E) Too many indirect identifiers**

The number of names that can be specified in a file of the indirect memory function exceeds the limit of 256.

**C2370 (E) Illegal #pragma regsave/noregsave declaration**

The **#pragma regsave** or **#pragma noregsave** declaration is illegal.

**C2371 (E) Function "function name" in #pragma regsave/noregsave function already declared**

The function specified by **#pragma regsave** or **#pragma noregsave** has already been declared as a function.

**C2372 (E) Illegal #pragma regsave/noregsave function type**

The function specified by **#pragma regsave** or **#pragma noregsave** has been declared or defined as a type other than a function.

**C2380 (E) Illegal #pragma inline/inline\_asm declaration**

The **#pragma inline** or **#pragma inline\_asm** declaration is illegal.

**C2381 (E) Function "function name" in #pragma inline/inline\_asm function already declared**

The function specified by **#pragma inline** or **#pragma inline\_asm** has already been declared as a function.

**C2382 (E) Illegal #pragma inline/inline\_asm function type**

The function specified by **#pragma inline** or **#pragma inline\_asm** has been declared or defined as a type other than a function.

**C2383 (E) #pragma inline\_asm ignored**

**#pragma inline\_asm** has been specified when the object is a relocatable object program.

**C2390 (E) Illegal #pragma global\_register declaration**

The **#pragma global\_register** declaration is illegal.

**C2391 (E) Variable "variable name" in #pragma global\_register already declared**

The variable specified by **#pragma global\_register** has already been declared as a variable.



**C2392 (E) Illegal #pragma global\_register symbol type**

The variable specified by **#pragma global\_register** has been declared as a type other than a variable.

**C2393 (E) Illegal register**

The register name specified by **#pragma global\_register** is illegal, or one register is specified more than once.

**C2400 (E) Illegal character "character"**

An illegal character is found.

**C2401 (E) Incomplete character constant**

An end of line indicator is detected in the middle of a character constant.

**C2402 (E) Incomplete string**

An end of line indicator is detected in the middle of a string literal.

**C2403 (E) EOF in comment**

An end of file indicator is detected in the middle of a comment.

**C2404 (E) Illegal character code "character code"**

An illegal character code is found.

**C2405 (E) Null character constant**

There are no characters in a character constant (i.e., no characters are specified between two quotation marks).

**C2407 (E) Incomplete logical line**

A backslash (\) or a backslash followed by an end of line indicator (\ (RET) ) is specified as the last character in a non-empty source file.

**C2408 (E) Comment nest too deep**

The nesting level of the comment exceeds 255.

**C2410 (E) Illegal #pragma entry declaration**

A syntax error has been found in the **#pragma entry** declaration.

**C2411 (E) Function "function name" in #pragma entry already declared**

Before the **#pragma entry** declaration, a symbol with the same name or a pragma is specified.

**C2412 (E) Illegal #pragma entry function type**

The specified symbol is not a function.

### **C2413 (E) Multiple #pragma entry declaration**

Multiple **#pragma entry** declarations exist.

### **C2420 (E) Illegal #pragma pack/unpack declaration**

A syntax error has been found in the **#pragma pack** or **#pragma unpack** declaration.

### **C2440 (E) Illegal #pragma stacksize declaration**

A syntax error has been found in the **#pragma stacksize** declaration.

### **C2441 (E) Multiple #pragma stacksize declaration**

Multiple **#pragma stacksize** declarations exist.

### **C2442 (E) Stack size overflow**

The stack size specified by **#pragma stacksize** is too large.

### **C2450 (E) Illegal #pragma option declaration**

An illegal **#pragma option** is declared.

### **C2460 (E) Pragma kind mismatch**

A **#pragma** type mismatch is detected in declarations.

### **C2470 (E) Illegal #pragma bit\_order declaration**

An illegal **#pragma bit\_order** is declared.

### **C2480 (E) Illegal #pragma address declaration**

An illegal **#pragma address** is specified.

### **C2481 (E) Variable "variable name" in #pragma address already declared**

Declaration preceded the **#pragma address** directive for **"variable name"**.

### **C2482 (E) Illegal #pragma address symbol type**

A symbolic name other than that of a variable was specified in a **#pragma address** directive.

### **C2483 (E) Illegal address in #pragma address**

- (1) An odd address was specified for a variable or structure that requires an even-address boundary.
- (2) The same address was specified for more than one variable or there is an overlap between address ranges occupied by variables.
- (3) There is an overlap between address ranges specified in two **#pragma address** directives.

### **C2500 (E) Illegal token "tokens"**

An illegal token sequence is used.

**C2501 (E) Division by zero**

An integer is divided by zero in a constant expression.

**C2510 (E) Missing {**

"{" that starts the `__asm` block is not found.

**C2511 (E) Illegal mnemonic**

Illegal mnemonics are used.

**C2512 (E) Member reference required for "offset"**

An **offset** operator is used for a purpose other than referencing members.

**C2513 (E) Number of operands mismatch**

The number of operands is illegal.

**C2514 (E) Illegal addressing mode**

An illegal addressing mode is specified in an operand.

**C2515 (E) Illegal register list**

An illegal specification is made in the register list.

**C2516 (E) Constant required**

No constant is specified.

**C2517 (E) Illegal value in operand**

A value outside the range is specified in an operand.

**C2518 (E) Invalid delay slot instruction**

An illegal instruction is located in the delay slot.

**C2600 (E) #error : "string literal"**

An error message specified by the **#error** string literal is output to the list file if the **nolist** option is not specified.

**C2801 (E) Illegal parameter type in intrinsic function**

There are different parameter types in an intrinsic function.

**C2802 (E) Parameter out of range in intrinsic function**

A parameter exceeds the range that can be specified in an intrinsic function.

**C2803 (E) Usage for intrinsic function is wrong**

An intrinsic function is erroneously used.

**C3000 (F) Statement nest too deep**

The nesting levels of an **if**, **while**, **for**, **do**, or **switch** statement exceeds 256.

**C3006 (F) Too many parameters**

The number of parameters in either a function declaration or a function call exceeds 63.

**C3007 (F) Too many macro parameters**

The number of parameters in a macro definition or a macro call exceeds 63.

**C3008 (F) Line too long**

After a macro expansion, the length of a line exceeds 16384 characters.

**C3009 (F) String literal too long**

The length of a string literal exceeds 32767 characters. The length of a string literal equals to the number of bytes when linking string literals specified continuously. The length of the string literal is not the length in the source program but the number of bytes included in the string literal data. Escape sequence is counted as one character.

**C3013 (F) Too many switches**

The number of **switch** statements exceeds 2048.

**C3014 (F) For nest too deep**

The nesting level of a **for** statement exceeds 128.

**C3017 (F) Too many case labels**

The number of **case** labels in one **switch** statement exceeds 511.

**C3018 (F) Too many goto labels**

The number of **goto** labels defined in one function exceeds 511.

**C3019 (F) Cannot open source file "file name"**

A source file cannot be opened.

**C3020 (F) Source file input error**

A source or include file cannot be read.

**C3021 (F) Memory overflow**

The compiler cannot allocate sufficient memory to compile the program.

**C3022 (F) Switch nest too deep**

The nesting level of a **switch** statement exceeds 128.

**C3023 (F) Type nest too deep**

The number of types (pointer, array, and function) that qualify the basic type exceeds 16.

**C3024 (F) Array dimension too deep**

The number of dimensions in an array exceeds six.

**C3025 (F) Source file not found**

A source file name is not specified in the command line.

**C3026 (F) Expression too complex**

An expression is too complex.

**C3027 (F) Source file too complex**

The nesting level of statements in the program is too deep or an expression is too complex.

**C3030 (F) Too many compound statements**

The number of compound statements in one function exceeds 2048.

**C3031 (F) Data size overflow**

The size of an array or a structure exceeds the limit. The limit for each CPU/operating mode is as follows:

- 65535 for **H8SXN**, **2600n**, **2000n**, **300hn**, or **300**
- 32767 for **H8SXA** with **ptr16** option, **H8SXX** with **ptr16** option, or **H8SXM**
- 1048575 for **H8SXA:20**, **2600a:20**, **2000a:20**, or **300ha:20**
- 16777215 for **H8SXA:24**, **2600a:24**, **2000a:24**, or **300ha:24**
- 268435455 for **H8SXA:28**, **H8SXM:28**, **2600a:28**, or **2000a:28**
- 4294967295 for **H8SXA:32**, **H8SXM:32**, **2600a:32**, or **2000a:32**

**C3034 (F) Invalid file name "file name"**

The specification of the file name is invalid.

**C3200 (F) Object size overflow**

The size of the object program exceeds the memory limit. The limit for each CPU/operating mode is as follows:

- 65535 for **H8SXN**, **2600n**, **2000n**, **300hn**, or **300**
- 1048575 for **H8SXA:20**, **H8SXM:20**, **2600a:20**, **2000a:20**, or **300ha:20**
- 16777215 for **H8SXA:24**, **H8SXM:24**, **2600a:24**, **2000a:24**, or **300ha:24**
- 268435455 for **H8SXA:28**, **H8SXM:28**, **2600a:28**, or **2000a:28**
- 4294967295 for **H8SXA:32**, **H8SXM:32**, **2600a:32**, or **2000a:32**

### **C3201 (F) Object data size overflow**

The data size exceeds the memory limit. The limit for each CPU/operating mode is as follows:

- 65535 for **H8SXN, H8SXM, 2600n, 2000n, 300hn, or 300**
- 65535 for **H8SXA with ptr16 option, or H8SXX with ptr16 option**
- 1048575 for **H8SXA:20, 2600a:20, 2000a:20, or 300ha:20**
- 16777215 for **H8SXA:24, 2600a:24, 2000a:24, or 300ha:24**
- 268435455 for **H8SXA:28, H8SXM:28, 2600a:28, or 2000a:28**
- 4294967295 for **H8SXA:32, H8SXM:32, 2600a:32, or 2000a:32**

### **C3202 (F) Illegal stack access**

The local variable and temporary area, and the register save area are placed at an address that exceeds the limit value for the stack pointer (SP) or frame pointer (FP), or the parameter area is placed at an address that exceeds the limit value for the SP or FP. The offset limit from an SP or FP for each CPU/operating mode is as follows:

- 32767 for **H8SXN, H8SXM, 2600n, 2000n, 300hn, or 300**
- 32767 for **H8SXA with ptr16 option, or H8SXX with ptr16 option**
- 524287 for **H8SXA:20, 2600a:20, 2000a:20, or 300ha:20**
- 8388607 for **H8SXA:24, 2600a:24, 2000a:24, or 300ha:24**
- 134217727 for **H8SXA:28, H8SXM:28, 2600a:28, or 2000a:28**
- 2147483647 for **H8SXA:32, H8SXM:32, 2600a:32, or 2000a:32**

### **C3300 (F) Cannot open internal file**

An intermediate file internally used by the compiler cannot be opened.

### **C3301 (F) Cannot close internal file**

An intermediate file internally generated by the compiler cannot be closed. Check that the intermediate file generated by the compiler is not being used.

### **C3302 (F) Cannot input internal file**

An intermediate file internally generated by the compiler cannot be read. Check that the intermediate file generated by the compiler is not being used.

### **C3303 (F) Cannot output internal file**

An intermediate file internally generated by the compiler cannot be written to. Increase the disk size.

### **C3304 (F) Cannot delete internal file**

An intermediate file internally generated by the compiler cannot be deleted. Check that the intermediate file generated by the compiler is not being used.

**C3305 (F) Invalid command parameter "option"**

An invalid compiler option is specified.

**C3306 (F) Interrupt in compilation**

An interrupt generated by (cntl)+C keys (from a standard input terminal) is detected during compilation.

**C3307 (F) Compiler version mismatch in "file name"**

The file version specified by the "file name" in the compiler does not match other file versions. Refer to the Install Guide for the installation procedure, and reinstall the compiler.

**C3320 (F) Command parameter buffer overflow**

The command line specification exceeds 4096 characters.

**C3322 (F) Lacking cpu specification**

The CPU/operating mode is not specified. Specify the CPU/operating mode with the **cpu** option or with environment variable H38CPU.

**C3323 (F) Illegal environment specified "environment variable"**

An error has been found in the specification of the environment variable (CH38TMP, H38CPU) used by the compiler.

**C3324 (F) Cannot open subcommand file "file name"**

The subcommand file cannot be opened.

**C3325 (F) Cannot close subcommand file**

The subcommand file cannot be closed. Check that the subcommand file is not being used.

**C3326 (F) Cannot input subcommand file**

The subcommand file cannot be read.

**C3327 (F) Cannot find "file name"**

The cross-software executable file cannot be found. Check whether the file name or path name is correct.

**C4xxx (–) Internal error**

An internal error occurred during compilation. Report the error to your sales agency.

**C5003 (F) #include file "file name" includes itself****C5004 (F) Out of memory****C5005 (F) Could not open source file "name"**

- C5006 (E) Comment unclosed at end of file**
- C5007 (E) (I) Unrecognized token**
- C5008 (E) (I) Missing closing quote**
- C5009 (I) Nested comment is not allowed**
- C5010 (E) "#" not expected here**
- C5011 (E) Unrecognized preprocessing directive**
- C5012 (E) Parsing restarts here after previous syntax error**
- C5013 (F) (E) Expected a file name**
- C5014 (E) Extra text after expected end of preprocessing directive**
- C5016 (F) "name" is not a valid source file name**
- C5017 (E) Expected a "]"**
- C5018 (E) Expected a ")"**
- C5019 (E) Extra text after expected end of number**
- C5020 (E) Identifier "name" is undefined**
- C5021 (W) Type qualifiers are meaningless in this declaration**
- C5022 (E) Invalid hexadecimal number**
- C5024 (E) Invalid octal digit**
- C5025 (E) Quoted string should contain at least one character**



- C5026 (E) Too many characters in character constant**
- C5027 (W) Character value is out of range**
- C5028 (E) Expression must have a constant value**
- C5029 (E) Expected an expression**
- C5030 (E) Floating constant is out of range**
- C5031 (E) Expression must have integral type**
- C5032 (E) Expression must have arithmetic type**
- C5033 (E) Expected a line number**
- C5034 (E) Invalid line number**
- C5035 (F) #error directive: "line number"**
- C5036 (E) The #if for this directive is missing**
- C5037 (E) The #endif for this directive is missing**
- C5038 (W) Directive is not allowed -- an #else has already appeared**
- C5039 (E) Division by zero**
- C5040 (E) Expected an identifier**
- C5041 (E) Expression must have arithmetic or pointer type**
- C5042 (E) Operand types are incompatible ("type 1" and "type 2")**
- C5044 (E) Expression must have pointer type**
- C5045 (W) #undef may not be used on this predefined name**

- C5046 (W) This predefined name may not be redefined**
- C5047 (W) Incompatible redefinition of macro "name" (declared at line "line number")**
- C5049 (E) Duplicate macro parameter name**
- C5050 (E) "##" may not be first in a macro definition**
- C5051 (E) "##" may not be last in a macro definition**
- C5052 (E) Expected a macro parameter name**
- C5053 (E) Expected a ":"**
- C5054 (W) Too few arguments in macro invocation**
- C5055 (W) Too many arguments in macro invocation**
- C5056 (E) Operand of sizeof may not be a function**
- C5057 (E) This operator is not allowed in a constant expression**
- C5058 (E) This operator is not allowed in a preprocessing expression**
- C5059 (E) Function call is not allowed in a constant expression**
- C5060 (E) This operator is not allowed in an integral constant expression**
- C5061 (W) Integer operation result is out of range**
- C5062 (W) Shift count is negative**
- C5063 (W) Shift count is too large**
- C5064 (W) Declaration does not declare anything**

- C5065 (E) Expected a ";"**
- C5066 (E) Enumeration value is out of "int" range**
- C5067 (E) Expected a "}"**
- C5068 (W) Integer conversion resulted in a change of sign**
- C5069 (W) Integer conversion resulted in truncation**
- C5070 (E) Incomplete type is not allowed**
- C5071 (E) Operand of sizeof may not be a bit field**
- C5075 (E) Operand of "\*" must be a pointer**
- C5077 (E) This declaration has no storage class or type specifier**
- C5079 (E) Expected a type specifier**
- C5080 (E) A storage class may not be specified here**
- C5081 (E) More than one storage class may not be specified**
- C5083 (W) Type qualifier specified more than once**
- C5084 (E) Invalid combination of type specifiers**
- C5085 (E) Invalid storage class for a parameter**
- C5086 (E) Invalid storage class for a function**
- C5087 (E) A type specifier may not be used here**
- C5088 (E) Array of functions is not allowed**
- C5089 (E) Array of void is not allowed**

- C5090 (E) Function returning function is not allowed**
- C5091 (E) Function returning array is not allowed**
- C5093 (E) Function type may not come from a typedef**
- C5094 (E) The size of an array must be greater than zero**
- C5095 (E) Array is too large**
- C5097 (E) A function may not return a value of this type**
- C5098 (E) An array may not have elements of this type**
- C5100 (E) Duplicate parameter name**
- C5101 (E) "name" has already been declared in the current scope**
- C5103 (E) Class is too large**
- C5105 (E) Invalid size for bit field**
- C5106 (E) Invalid type for a bit field**
- C5107 (E) Zero-length bit field must be unnamed**
- C5108 (W) Signed bit field of length 1**
- C5109 (E) Expression must have (pointer-to-) function type**
- C5110 (E) Expected either a definition or a tag name**
- C5111 (I) Statement is unreachable**
- C5112 (E) Expected "while"**

- C5114 (E) Entity-kind "name" was referenced but not defined**
- C5115 (E) A continue statement may only be used within a loop**
- C5116 (E) A break statement may only be used within a loop or switch**
- C5117 (W) non-void entity-kind "name" should return a value**
- C5118 (E) A void function may not return a value**
- C5119 (E) Cast to type "type" is not allowed**
- C5120 (E) Return value type does not match the function type**
- C5121 (E) A case label may only be used within a switch**
- C5122 (E) A default label may only be used within a switch**
- C5123 (E) Case label value has already appeared in this switch**
- C5124 (E) Default label has already appeared in this switch**
- C5125 (E) Expected a "("**
- C5126 (E) Expression must be an lvalue**
- C5127 (E) Expected a statement**
- C5128 (I) Loop is not reachable from preceding code**
- C5129 (E) A block-scope function may only have extern storage class**
- C5130 (E) Expected a "{"**
- C5131 (E) Expression must have pointer-to-class type**
- C5132 (E) Expression must have pointer-to-struct-or-union type**

- C5133 (E) Expected a member name**
- C5134 (E) Expected a field name**
- C5135 (E) Entity-kind "name" has no member "member name"**
- C5136 (E) Entity-kind "name" has no field "field name"**
- C5137 (E) Expression must be a modifiable lvalue**
- C5139 (E) Taking the address of a bit field is not allowed**
- C5140 (E) Too many arguments in function call**
- C5142 (E) Expression must have pointer-to-object type**
- C5143 (F) Program too large or complicated to compile**
- C5144 (E) A value of type "type 1" cannot be used to initialize an entity of type "type 2"**
- C5145 (E) Entity-kind "name" may not be initialized**
- C5146 (E) Too many initializer values**
- C5147 (E) Declaration is incompatible with "name" (declared at line "line number")**
- C5148 (E) Entity-kind "name" has already been initialized**
- C5149 (E) A global-scope declaration may not have this storage class**
- C5150 (E) A type name may not be redeclared as a parameter**
- C5151 (E) A typedef name may not be redeclared as a parameter**
- C5153 (E) Expression must have class type**

- C5154 (E) Expression must have struct or union type**
- C5157 (E) Expression must be an integral constant expression**
- C5158 (E) Expression must be an lvalue or a function designator**
- C5159 (E) Declaration is incompatible with previous "name" (declared at line "line number")**
- C5160 (E) Name conflicts with previously used external name "name"**
- C5161 (I) Unrecognized #pragma**
- C5163 (F) Could not open temporary file "name"**
- C5164 (F) Name of directory for temporary files is too long ("name")**
- C5165 (E) Too few arguments in function call**
- C5166 (E) Invalid floating constant**
- C5167 (E) Argument of type "type 1" is incompatible with parameter of type "type 2"**
- C5168 (E) A function type is not allowed here**
- C5169 (E) Expected a declaration**
- C5170 (W) Pointer points outside of underlying object**
- C5171 (E) Invalid type conversion**
- C5172 (I) External/internal linkage conflict with previous declaration**
- C5173 (E) Floating-point value does not fit in required integral type**
- C5174 (I) Expression has no effect**

- C5175 (W) Subscript out of range**
- C5177 (W) Entity-kind "name" was declared but never referenced**
- C5179 (W) Right operand of "%" is zero**
- C5182 (F) Could not open source file "name" (no directories in search list)**
- C5183 (E) Type of cast must be integral**
- C5184 (E) Type of cast must be arithmetic or pointer**
- C5185 (I) Dynamic initialization in unreachable code**
- C5186 (W) Pointless comparison of unsigned integer with zero**
- C5187 (I) Use of "=" where "= =" may have been intended**
- C5189 (F) Error while writing "file name" file**
- C5191 (W) Type qualifier is meaningless on cast type**
- C5192 (W) Unrecognized character escape sequence**
- C5193 (I) Zero used for undefined preprocessing identifier**
- C5219 (F) Error while deleting file "file name"**
- C5221 (W) Floating-point value does not fit in required floating-point type**
- C5224 (W) The format string requires additional arguments**
- C5225 (W) The format string ends before this argument**
- C5226 (W) Invalid format string conversion**
- C5229 (W) Bit field cannot contain all values of the enumerated type**



- C5235 (E) Variable "name" was declared with a never-completed type**
- C5236 (W) (I) Controlling expression is constant**
- C5237 (I) Selector expression is constant**
- C5238 (E) Invalid specifier on a parameter**
- C5239 (E) Invalid specifier outside a class declaration**
- C5240 (E) Duplicate specifier in declaration**
- C5241 (E) A union is not allowed to have a base class**
- C5242 (E) Multiple access control specifiers are not allowed**
- C5243 (E) Class or struct definition is missing**
- C5244 (E) Qualified name is not a member of class "type" or its base classes**
- C5245 (E) A nonstatic member reference must be relative to a specific object**
- C5246 (E) A nonstatic data member may not be defined outside its class**
- C5247 (E) Entity-kind "name" has already been defined**
- C5248 (E) Pointer to reference is not allowed**
- C5249 (E) Reference to reference is not allowed**
- C5250 (E) Reference to void is not allowed**
- C5251 (E) Array of reference is not allowed**
- C5252 (E) Reference entity-kind "name" requires an initializer**

- C5253 (E) Expected a ","**
- C5254 (E) Type name is not allowed**
- C5255 (E) Type definition is not allowed**
- C5256 (E) Invalid redeclaration of type name "name" (declared at line "line number")**
- C5257 (E) Const entity-kind "name" requires an initializer**
- C5258 (E) "this" may only be used inside a nonstatic member function**
- C5259 (E) Constant value is not known**
- C5261 (I) Access control not specified ("name" by default)**
- C5262 (E) Not a class or struct name**
- C5263 (E) Duplicate base class name**
- C5264 (E) Invalid base class**
- C5265 (E) Entity-kind "name" is inaccessible**
- C5266 (E) "name" is ambiguous**
- C5269 (E) Implicit conversion to inaccessible base class "type" is not allowed**
- C5274 (E) Improperly terminated macro invocation**
- C5276 (E) Name followed by "::" must be a class or namespace name**
- C5277 (E) Invalid friend declaration**
- C5278 (E) A constructor or destructor may not return a value**
- C5279 (E) Invalid destructor declaration**

- C5280 (E) (W) Declaration of a member with the same name as its class**
- C5281 (E) Global-scope qualifier (leading "::") is not allowed**
- C5282 (E) The global scope has no "name"**
- C5283 (E) Qualified name is not allowed**
- C5284 (W) NULL reference is not allowed**
- C5285 (E) Initialization with "{...}" is not allowed for object of type "type"**
- C5286 (E) Base class "type" is ambiguous**
- C5287 (E) Derived class "type" contains more than one instance of class "type"**
- C5288 (E) Cannot convert pointer to base class "type 1" to pointer to derived class "type 2" -- base class is virtual**
- C5289 (E) No instance of constructor "name" matches the argument list**
- C5290 (E) Copy constructor for class "type" is ambiguous**
- C5291 (E) No default constructor exists for class "type"**
- C5292 (E) "name" is not a nonstatic data member or base class of class "type"**
- C5293 (E) Indirect nonvirtual base class is not allowed**
- C5294 (E) Invalid union member -- class "type" has a disallowed member function**
- C5297 (E) Expected an operator**
- C5298 (E) Inherited member is not allowed**
- C5299 (E) Cannot determine which instance of entity-kind "name" is intended**

- C5300 (E) A pointer to a bound function may only be used to call the function**
- C5302 (E) Entity-kind "name" has already been defined**
- C5304 (E) No instance of entity-kind "name" matches the argument list**
- C5305 (E) Type definition is not allowed in function return type declaration**
- C5306 (E) Default argument not at end of parameter list**
- C5307 (E) Redefinition of default argument**
- C5308 (E) More than one instance of entity-kind "name" matches the argument list:**
- C5309 (E) More than one instance of constructor "name" matches the argument list:**
- C5310 (E) Default argument of type "type 1" is incompatible with parameter of type "type 2"**
- C5311 (E) Cannot overload functions distinguished by return type alone**
- C5312 (E) No suitable user-defined conversion from "type 1" to "type 2" exists**
- C5313 (E) Type qualifier is not allowed on this function**
- C5314 (E) Only nonstatic member functions may be virtual**
- C5315 (E) The object has type qualifiers that are not compatible with the member function**
- C5316 (E) Program too large to compile (too many virtual functions)**
- C5317 (E) Return type is not identical to nor covariant with return type "type" of overridden virtual function entity-kind "name"**
- C5318 (E) Override of virtual entity-kind "name" is ambiguous**

- C5319 (E) Pure specifier ("= 0") allowed only on virtual functions**
- C5320 (E) Badly-formed pure specifier (only "= 0" is allowed)**
- C5321 (E) Data member initializer is not allowed**
- C5322 (E) Object of abstract class type "type" is not allowed:**
- C5323 (E) Function returning abstract class "type" is not allowed:**
- C5324 (I) Duplicate friend declaration**
- C5325 (E) Inline specifier allowed on function declarations only**
- C5326 (E) "inline" is not allowed**
- C5327 (E) Invalid storage class for an inline function**
- C5328 (E) Invalid storage class for a class member**
- C5329 (E) Local class member entity-kind "name" requires a definition**
- C5330 (E) Entity-kind "name" is inaccessible**
- C5332 (E) Class "type" has no copy constructor to copy a const object**
- C5333 (E) Defining an implicitly declared member function is not allowed**
- C5334 (E) Class "type" has no suitable copy constructor**
- C5335 (E) Linkage specification is not allowed**
- C5336 (E) Unknown external linkage specification**
- C5337 (E) Linkage specification is incompatible with previous "name" (declared at line "line number")**

- C5338 (E) More than one instance of overloaded function "name" has "C" linkage
- C5339 (E) Class "type" has more than one default constructor
- C5341 (E) "operator" must be a member function
- C5342 (E) Operator may not be a static member function
- C5343 (E) No arguments allowed on user-defined conversion
- C5344 (E) Too many parameters for this operator function
- C5345 (E) Too few parameters for this operator function
- C5346 (E) Nonmember operator requires a parameter with class type
- C5347 (E) Default argument is not allowed
- C5348 (E) More than one user-defined conversion from "type 1" to "type 2" applies:
- C5349 (E) No operator "operator" matches these operands
- C5350 (E) More than one operator "operator" matches these operands:
- C5351 (E) First parameter of allocation function must be of type "size\_t"
- C5352 (E) Allocation function requires "void \*" return type
- C5353 (E) Deallocation function requires "void" return type
- C5354 (E) First parameter of deallocation function must be of type "void \*"
- C5356 (E) Type must be an object type
- C5357 (E) Base class "type" has already been initialized
- C5359 (E) Entity-kind "name" has already been initialized

- C5360 (E) Name of member or base class is missing**
- C5363 (E) Invalid anonymous union -- nonpublic member is not allowed**
- C5364 (E) Invalid anonymous union -- member function is not allowed**
- C5365 (E) Anonymous union at global or namespace scope must be declared static**
- C5366 (E) Entity-kind "name" provides no initializer for:**
- C5367 (E) Implicitly generated constructor for class "type" cannot initialize:**
- C5368 (W) Entity-kind "name" defines no constructor to initialize the following:**
- C5369 (E) Entity-kind "name" has an uninitialized const or reference member**
- C5370 (W) Entity-kind "name" has an uninitialized const field**
- C5371 (E) Class "type" has no assignment operator to copy a const object**
- C5372 (E) Class "type" has no suitable assignment operator**
- C5373 (E) Ambiguous assignment operator for class "type"**
- C5375 (E) Declaration requires a typedef name**
- C5377 (E) "virtual" is not allowed**
- C5378 (E) "static" is not allowed**
- C5380 (E) Expression must have pointer-to-member type**
- C5381 (I) Extra ";" ignored**
- C5382 (W) Nonstandard member constant declaration (standard form is a static const integral member)**

- C5384 (E) No instance of overloaded "name" matches the argument list**
- C5386 (E) No instance of entity-kind "name" matches the required type**
- C5388 (E) "operator->" for class "type 1" returns invalid type "type 2"**
- C5389 (E) A cast to abstract class "type" is not allowed:**
- C5391 (E) A new-initializer may not be specified for an array**
- C5392 (E) Member function "name" may not be redeclared outside its class**
- C5393 (E) Pointer to incomplete class type is not allowed**
- C5394 (E) Reference to local variable of enclosing function is not allowed**
- C5397 (E) Implicitly generated assignment operator cannot copy:**
- C5399 (I) Entity-kind "name" has an operator newxxxx () but no default operator deletexxxx ()**
- C5400 (I) Entity-kind "name" has a default operator deletexxxx () but no operator newxxxx ()**
- C5401 (E) Destructor for base class "type" is not virtual**
- C5403 (E) Entity-kind "name" has already been declared**
- C5404 (E) Function "main" may not be declared inline**
- C5405 (E) Member function with the same name as its class must be a constructor**
- C5407 (E) A destructor may not have parameters**
- C5408 (E) Copy constructor for class "type 1" may not have a parameter of type "type2"**
- C5409 (E) Entity-kind "name" returns incomplete type "type"**



- C5410 (E) Protected entity-kind "name" is not accessible through a "type" pointer or object**
- C5411 (E) A parameter is not allowed**
- C5412 (E) An "asm" declaration is not allowed here**
- C5413 (E) No suitable conversion function from "type 1" to "type 2" exists**
- C5414 (W) Delete of pointer to incomplete class**
- C5415 (E) No suitable constructor exists to convert from "type 1" to "type 2"**
- C5416 (E) More than one constructor applies to convert from "type 1" to "type 2":**
- C5417 (E) More than one conversion function from "type 1" to "type 2" applies:**
- C5418 (E) More than one conversion function from "type" to a built-in type applies:**
- C5424 (E) A constructor or destructor may not have its address taken**
- C5427 (E) Qualified name is not allowed in member declaration**
- C5429 (E) The size of an array in "new" must be non-negative**
- C5430 (W) Returning reference to local temporary**
- C5432 (E) "enum" declaration is not allowed**
- C5433 (E) Qualifiers dropped in binding reference of type "type 1" to initializer of type "type 2"**
- C5434 (E) A reference of type "type 1" (not const-qualified) cannot be initialized with a value of type "type 2"**
- C5435 (E) A pointer to function may not be deleted**

- C5436 (E) Conversion function must be a nonstatic member function**
- C5437 (E) Template declaration is not allowed here**
- C5438 (E) Expected a "<"**
- C5439 (E) Expected a ">"**
- C5440 (E) Template parameter declaration is missing**
- C5441 (E) Argument list for entity-kind "name" is missing**
- C5442 (E) Too few arguments for entity-kind "name"**
- C5443 (E) Too many arguments for entity-kind "name"**
- C5445 (E) Entity-kind "name 1" is not used in declaring the parameter types of entity-kind "name 2"**
- C5449 More than one instance of entity-kind "name" matches the required type**
- C5452 (E) Return type may not be specified on a conversion function**
- C5456 (E) Excessive recursion at instantiation of entity-kind "name"**
- C5457 (E) "name" is not a function or static data member**
- C5458 (E) Argument of type "type 1" is incompatible with template parameter of type "type 2"**
- C5459 (E) Initialization requiring a temporary or conversion is not allowed**
- C5461 (E) Initial value of reference to non-const must be an lvalue**
- C5463 (E) "template" is not allowed**
- C5464 (E) "type" is not a class template**

- C5466 (E) "main" is not a valid name for a function template**
- C5467 (E) Invalid reference to entity-kind "name" (union/nonunion mismatch)**
- C5468 (E) A template argument may not reference a local type**
- C5469 (E) Tag kind of "name 1" is incompatible with declaration of entity-kind "name 2" (declared at line "line number")**
- C5470 (E) The global scope has no tag named "name"**
- C5471 (E) Entity-kind "name 1" has no tag member named "name 2"**
- C5473 (E) Entity-kind "name" may be used only in pointer-to-member declaration**
- C5475 (E) A template argument may not reference a non-external entity**
- C5476 (E) Name followed by "::~" must be a class name or a type name**
- C5477 (E) Destructor name does not match name of class "type"**
- C5478 (E) Type used as destructor name does not match type "type"**
- C5479 (I) Entity-kind "name" redeclared "inline" after being called**
- C5481 (E) Invalid storage class for a template declaration**
- C5484 (E) Invalid explicit instantiation declaration**
- C5485 (E) Entity-kind "name" is not an entity that can be instantiated**
- C5486 (E) Compiler generated entity-kind "name" cannot be explicitly instantiated**
- C5487 (E) Inline entity-kind "name" cannot be explicitly instantiated**
- C5488 (E) Pure virtual entity-kind "name" cannot be explicitly instantiated**

- C5489 (E) Entity-kind "name" cannot be instantiated -- no template definition was supplied**
- C5490 (E) Entity-kind "name" cannot be instantiated -- it has been explicitly specialized**
- C5493 (E) No instance of entity-kind "name" matches the specified type**
- C5496 (E) Template parameter "name" may not be redeclared in this scope**
- C5497 (W) Declaration of "name" hides template parameter**
- C5498 (E) Template argument list must match the parameter list**
- C5499 (E) Conversion function to convert from "type 1" to "type 2" is not allowed**
- C5500 (E) Extra parameter of postfix "operatorxxxx" must be of type "int"**
- C5501 (E) An operator name must be declared as a function**
- C5502 (E) Operator name is not allowed**
- C5503 (E) Entity-kind "name" cannot be specialized in the current scope**
- C5505 (E) Too few template parameters -- does not match previous declaration**
- C5506 (E) Too many template parameters -- does not match previous declaration**
- C5507 (E) Function template for operator delete (void \*) is not allowed**
- C5508 (E) Class template and template parameter may not have the same name**
- C5510 (E) A template argument may not reference an unnamed type**
- C5511 (E) Enumerated type is not allowed**

- C5512 (W) Type qualifier on a reference type is not allowed**
- C5513 (E) A value of type "type 1" cannot be assigned to an entity of type "type 2"**
- C5514 (W) Pointless comparison of unsigned integer with a negative constant**
- C5515 (E) Cannot convert to incomplete class "type"**
- C5516 (E) Const object requires an initializer**
- C5517 (E) Object has an uninitialized const or reference member**
- C5519 (E) Entity-kind "name" may not have a template argument list**
- C5520 (E) Initialization with "{...}" expected for aggregate object**
- C5521 (E) Pointer-to-member selection class types are incompatible ("type 1" and "type 2")**
- C5522 (W) Pointless friend declaration**
- C5526 (E) A parameter may not have void type**
- C5529 (E) This operator is not allowed in a template argument expression**
- C5530 (E) Try block requires at least one handler**
- C5531 (E) Handler requires an exception declaration**
- C5532 (E) Handler is masked by default handler**
- C5533 (E) Handler is potentially masked by previous handler for type "type"**
- C5534 (I) Use of a local type to specify an exception**
- C5535 (I) Redundant type in exception specification**

- C5536 (E) Exception specification is incompatible with that of previous entity-kind "name" (declared at line "line number"):**
- C5540 (E) Support for exception handling is disabled**
- C5541 (W) Omission of exception specification is incompatible with previous entity-kind "name" (declared at line "line number")**
- C5542 (F) Could not create instantiation request file "name"**
- C5543 (E) Non-arithmetic operation not allowed in nontype template argument**
- C5544 (E) Use of a local type to declare a nonlocal variable**
- C5545 (E) Use of a local type to declare a function**
- C5546 (E) Transfer of control bypasses initialization of:**
- C5548 (E) Transfer of control into an exception handler**
- C5549 (W) Entity-kind "name" is used before its value is set**
- C5550 (W) Entity-kind "name" was set but never used**
- C5551 (E) Entity-kind "name" cannot be defined in the current scope**
- C5552 (W) Exception specification is not allowed**
- C5553 (W) External/internal linkage conflict for entity-kind "name" (declared at line "line number")**
- C5554 (W) Entity-kind "name" will not be called for implicit or explicit conversions**
- C5555 (E) Tag kind of "name" is incompatible with template parameter of type "type"**
- C5556 (E) Function template for operator new (size\_t) is not allowed**
- C5558 (E) Pointer to member of type "type" is not allowed**

- C5559 (E) Ellipsis is not allowed in operator function parameter list**
- C5598 (E) A template parameter may not have void type**
- C5601 (E) A throw expression may not have void type**
- C5603 (E) Parameter of abstract class type "type" is not allowed:**
- C5604 (E) Array of abstract class "type" is not allowed:**
- C5610 (W) Entity-kind "name 1" does not match "name 2" -- virtual function override intended?**
- C5611 (W) Overloaded virtual function "name 1" is only partially overridden in entity-kind "name 2"**
- C5612 (E) Specific definition of inline template function must precede its first use**
- C5624 (E) "name" is not a type name**
- C5641 (F) "name" is not a valid directory**
- C5642 (F) Cannot build temporary file name**
- C5656 (E) Transfer of control into a try block**
- C5657 (W) Inline specification is incompatible with previous "name" (declared at line "line number")**
- C5658 (E) Closing brace of template definition not found**
- C5660 (E) Invalid packing alignment value**
- C5662 (W) Call of pure virtual function**
- C5663 (E) Invalid source file identifier string**

- C5664 (E)** A class template cannot be defined in a friend declaration
- C5673 (E)** A reference of type "type 1" cannot be initialized with a value of type "type 2"
- C5674 (E)** Initial value of reference to const volatile must be an lvalue
- C5678 (I)** Call of entity-kind "name" (declared at line "line number") cannot be inlined
- C5679 (I)** Entity-kind "name" cannot be inlined
- C5693 (E)** <typeinfo> must be included before typeid is used
- C5694 (E)** "name" cannot cast away const or other type qualifiers
- C5695 (E)** The type in a dynamic\_cast must be a pointer or reference to a complete class type, or void \*
- C5696 (E)** The operand of a pointer dynamic\_cast must be a pointer to a complete class type
- C5697 (E)** The operand of a reference dynamic\_cast must be an lvalue of a complete class type
- C5698 (E)** The operand of a runtime dynamic\_cast must have a polymorphic class type
- C5701 (E)** An array type is not allowed here
- C5702 (E)** Expected an "="
- C5703 (E)** Expected a declarator in condition declaration
- C5704 (E)** "name", declared in condition, may not be redeclared in this scope
- C5705 (E)** Default template arguments are not allowed for function templates
- C5706 (E)** Expected a ",", " or ">"



- C5707 (E) Expected a template parameter list**
- C5708 (W) Incrementing a bool value is deprecated**
- C5709 (E) bool type is not allowed**
- C5710 (E) Offset of base class "name 1" within class "name 2" is too large**
- C5711 (E) Expression must have bool type (or be convertible to bool)**
- C5717 (E) The type in a const\_cast must be a pointer, reference, or pointer to member to an object type**
- C5718 (E) A const\_cast can only adjust type qualifiers; it cannot change the underlying type**
- C5719 (E) mutable is not allowed**
- C5720 (W) Redclaration of entity-kind "name" is not allowed to alter its access**
- C5722 (W) Use of alternative token "<:" appears to be unintended**
- C5723 (W) Use of alternative token "%:" appears to be unintended**
- C5724 (E) namespace definition is not allowed**
- C5725 (E) Name must be a namespace name**
- C5726 (E) Namespace alias definition is not allowed**
- C5727 (E) namespace-qualified name is required**
- C5728 (E) A namespace name is not allowed**
- C5730 (E) Entity-kind "name" is not a class template**
- C5732 (E) Allocation operator may not be declared in a namespace**

- C5733 (E) Deallocation operator may not be declared in a namespace**
- C5734 (E) Entity-kind "name 1" conflicts with using-declaration of entity-kind "name 2"**
- C5735 (E) Using-declaration of entity-kind "name 1" conflicts with entity-kind "name 2" (declared at line "line number")**
- C5737 (W) Using-declaration ignored -- it refers to the current namespace**
- C5738 (E) A class-qualified name is required**
- C5741 (W) Using-declaration of entity-kind "name" ignored**
- C5742 (E) Entity-kind "name 1" has no actual member "name 2"**
- C5750 (E) Entity-kind "name" (declared at line "line number") was used before its template was declared**
- C5751 (E) Static and nonstatic member functions with same parameter types cannot be overloaded**
- C5752 (E) No prior declaration of entity-kind "name"**
- C5753 (E) A template-id is not allowed**
- C5754 (E) A class-qualified name is not allowed**
- C5755 (E) Entity-kind "name" may not be redeclared in the current scope**
- C5756 (E) Qualified name is not allowed in namespace member declaration**
- C5757 (E) Entity-kind "name" is not a type name**
- C5761 (E) Typename may only be used within a template**

**C5766 (W) Exception specification for virtual entity-kind "name 1" is incompatible with that of overridden entity-kind "name 2"**

**C5767 (W) Conversion from pointer to smaller integer**

**C5768 (W) Exception specification for implicitly declared virtual entity-kind "name 1" is incompatible with that of overridden entity-kind "name 2"**

**C5771 (E) "explicit" is not allowed**

**C5772 (E) Declaration conflicts with "name" (reserved class name)**

**C5773 (E) Only "(" is allowed as initializer for array entity-kind "name"**

**C5774 (E) "virtual" is not allowed in a function template declaration**

**C5775 (E) Invalid anonymous union -- class member template is not allowed**

**C5776 (E) Template nesting depth does not match the previous declaration of entity-kind "name"**

**C5777 (E) This declaration cannot have multiple "template <...>" clauses**

**C5779 (E) "name", declared in for-loop initialization, may not be redeclared in this scope**

**C5782 (E) Definition of virtual entity-kind "name" is required here**

**C5784 (E) A storage class is not allowed in a friend declaration**

**C5785 (E) Template parameter list for "name" is not allowed in this declaration**

**C5786 (E) Entity-kind "name" is not a valid member class or function template**

**C5787 (E) Not a valid member class or function template declaration**

**C5788 (E) A template declaration containing a template parameter list may not be followed by an explicit specialization declaration**

- C5789 (E) Explicit specialization of entity-kind "name 1" must precede the first use of entity-kind "name 2"**
- C5790 (E) Explicit specialization is not allowed in the current scope**
- C5791 (E) Partial specialization of entity-kind "name" is not allowed**
- C5792 (E) Entity-kind "name" is not an entity that can be explicitly specialized**
- C5793 (E) Explicit specialization of entity-kind "name" must precede its first use**
- C5794 (W) Template parameter "template" may not be used in an elaborated type specifier**
- C5795 (E) Specializing entity-kind "name" requires "template<>" syntax**
- C5800 (E) This declaration may not have extern "C" linkage**
- C5801 (E) "name" is not a class or function template name in the current scope**
- C5802 (W) Specifying a default argument when redeclaring an unreferenced function template is nonstandard**
- C5803 (E) Specifying a default argument when redeclaring an already referenced function template is not allowed**
- C5804 (E) Cannot convert pointer to member of base class "type 1" to pointer to member of derived class "type 2" – base class is virtual**
- C5805 (E) Exception specification is incompatible with that of entity-kind "name" (declared at line "line number"):**
- C5806 (W) Omission of exception specification is incompatible with entity-kind "name" (declared at line "line number")**

- C5807 (E) The parse of this expression has changed between the point at which it appeared in the program and the point at which the expression was evaluated -- "typename" may be required to resolve the ambiguity**
- C5808 (E) Default-initialization of reference is not allowed**
- C5809 (E) Uninitialized entity-kind "name" has a const member**
- C5810 (E) Uninitialized base class "type" has a const member**
- C5811 (E) Const entity-kind "name" requires an initializer -- class "type" has no explicitly declared default constructor**
- C5812 (W) Const object requires an initializer -- class "type" has no explicitly declared default constructor**
- C5815 (I) Type qualifier on return type is meaningless**
- C5817 (E) Static data member declaration is not allowed in this class**
- C5818 (E) Template instantiation resulted in an invalid function declaration**
- C5822 (E) Invalid destructor name for type "type"**
- C5824 (E) Destructor reference is ambiguous -- both entity-kind "name 1" and entity-kind "name 2" could be used**
- C5825 (E) Virtual inline entity-kind "name" was never defined**
- C5826 (W) Entity-kind "name" was never referenced**
- C5827 (E) Only one member of a union may be specified in a constructor initializer list**
- C5831 (I) Support for placement delete is disabled**
- C5832 (E) No appropriate operator delete is visible**
- C5833 (E) Pointer or reference to incomplete type is not allowed**

- C5834 (E) Invalid partial specialization -- entity-kind "name" is already fully specialized**
- C5835 (E) Incompatible exception specifications**
- C5836 (W) Returning reference to local variable**
- C5837 (W) Omission of explicit type is nonstandard ("int" assumed)**
- C5838 (E) More than one partial specialization matches the template argument list of entity-kind "name"**
- C5840 (E) A template argument list is not allowed in a declaration of a primary template**
- C5841 (E) Partial specializations may not have default template arguments**
- C5842 (E) Entity-kind "name 1" is not used in template argument list of entity-kind "name 2"**
- C5843 (E) The type of partial specialization template parameter entity-kind "name" depends on another template parameter**
- C5844 (E) The template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter**
- C5845 (E) This partial specialization would have been used to instantiate entity-kind "name"**
- C5846 (E) This partial specialization would have been made the instantiation of entity-kind "name" ambiguous**
- C5847 (E) Expression must have integral or enum type**
- C5848 (E) Expression must have arithmetic or enum type**
- C5849 (E) Expression must have arithmetic, enum, or pointer type**

- C5850 (E) Type of cast must be integral or enum**
- C5851 (E) Type of cast must be arithmetic, enum, or pointer**
- C5852 (E) Expression must be a pointer to a complete object type**
- C5853 (E) A partial specialization of a member class template must be declared in the class of which it is a member**
- C5854 (E) A partial specialization nontype argument must be the name of a nontype parameter or a constant**
- C5855 (E) Return type is not identical to return type "type" of overridden virtual function entity-kind "name"**
- C5857 (E) A partial specialization of a class template must be declared in the namespace of which it is a member**
- C5858 (E) Entity-kind "name" is a pure virtual function**
- C5859 (E) Pure virtual entity-kind "name" has no overrider**
- C5861 (E) Invalid character in input line**
- C5862 (E) Function returns incomplete type "type"**
- C5864 (E) "name" is not a template**
- C5865 (E) A friend declaration may not declare a partial specialization**
- C5867 (W) Declaration of "size\_t" does not match the expected type "type"**
- C5868 (E) Space required between adjacent ">" delimiters of nested template argument lists (">>" is the right shift operator)**
- C5870 (W) Invalid multibyte character sequence**

- C5871 (E) Template instantiation resulted in unexpected function type of "type 1" (the meaning of a name may have changed since the template declaration -- the type of the template is "type 2")**
- C5873 (E) Non-integral operation not allowed in nontype template argument**
- C5875 (W) Embedded C++ does not support templates**
- C5876 (W) Embedded C++ does not support exception handling**
- C5877 (W) Embedded C++ does not support namespaces**
- C5878 (W) Embedded C++ does not support run-time type information**
- C5879 (W) Embedded C++ does not support the new cast syntax**
- C5880 (W) Embedded C++ does not support using-declarations**
- C5881 (W) Embedded C++ does not support "mutable"**
- C5882 (W) Embedded C++ does not support multiple or virtual inheritance**
- C5885 (E) "type 1" cannot be used to designate constructor for "type 2"**
- C5891 (E) An explicit template argument list is not allowed on this declaration**
- C5894 (E) Entity-kind "name" is not a template**
- C5896 (E) Expected a template argument**
- C5898 (E) Nonmember operator requires a parameter with class or enum type**
- C5900 (E) Using-declaration of entity-kind "name" is not allowed**
- C5901 (E) Qualifier of destructor name "type 1" does not match type "type 2"**
- C5902 (W) Type qualifier ignored**



- C5916 (E) Cannot convert pointer to member of derived class "type 1" to pointer to member of base class "type 2" – base class is virtual**
- C5919 (F) Invalid output file: "name"**
- C5920 (F) Cannot open output file: "name"**
- C5926 (F) Cannot open definition list file: "name"**
- C5928 (E) Incorrect use of va\_start**
- C5929 (E) Incorrect use of va\_arg**
- C5930 (E) Incorrect use of va\_end**
- C5935 (E) "typedef" may not be specified here**
- C5936 (W) Redclaration of entity-kind "name" alters its access**
- C5937 (E) A class or namespace qualified name is required**
- C5940 (W) Missing return statement at end of non-void entity-kind "name"**
- C5941 (W) Duplicate using-declaration of "name" ignored**
- C5946 (E) Name following "template" must be a member template**
- C5947 (E) Name following "template" must have a template argument list**
- C5952 (E) A template parameter may not have class type**
- C5953 (E) A default template argument cannot be specified on the declaration of a member of a class template**
- C5954 (E) A return statement is not allowed in a handler of a function try block of a constructor**

- C5959 (W) Declared size for bit field is larger than the size of the bit field type; truncated to "size" bits**
- C5960 (E) Type used as constructor name does not match type "type"**
- C5961 (W) Use of a type with no linkage to declare a variable with linkage**
- C5962 (W) Use of a type with no linkage to declare a function**
- C5963 (E) Return type may not be specified on a constructor**
- C5964 (E) Return type may not be specified on a destructor**
- C5965 (E) Incorrectly formed universal character name**
- C5966 (E) Universal character name specifies an invalid character**
- C5967 (E) A universal character name cannot designate a character in the basic character set**
- C5968 (E) This universal character is not allowed in an identifier**
- C5978 (E) A template friend declaration cannot be declared in a local class**
- C5979 (E) Ambiguous "?" operation: second operand of type "type 1" can be converted to third operand type "type 2", and vice versa**
- C5980 (E) Call of an object of a class type without appropriate operator ( ) or conversion functions to pointer-to-function type**
- C5982 (E) There is more than one way an object of type "type" can be called for the argument list**
- C5984 (W) Operator new and operator delete cannot be given internal linkage**
- C5985 (E) Storage class "mutable" is not allowed for anonymous unions**
- C5987 (E) Abstract class type "type" is not allowed as catch type:**

- C5988 (E) A qualified function type cannot be used to declare a nonmember function or a static member function**
- C5989 (E) A qualified function type cannot be used to declare a parameter**
- C5990 (E) Cannot create a pointer or reference to qualified function type**
- C5991 (W) Extra braces are nonstandard**
- C5994 (E) An empty template parameter list is not allowed in a template template parameter declaration**
- C5995 (E) Expected "class"**
- C5996 (E) The "class" keyword must be used when declaring a template template parameter**
- C5998 (E) A qualified name is not allowed for a friend declaration that is a function definition**
- C5999 (E) "type" is not compatible with "type"**
- C6000 (W) A storage class may not be specified here**
- C6006 (E) A template template parameter cannot have the same name as one of its template parameters**
- C6007 (W) "function name 1" is hidden by "function name 2" -- virtual function override intended?**
- C6008 (E) A parameter of a template template parameter cannot depend on the type of another template parameter**
- C6009 (E) "instance name" is not an entity that can be defined**

- C6010 (E) Destructor name must be qualified**
- C6013 (E) A qualified friend template declaration must refer to a specific previously declared template**
- C6018 (E) "class name" has no member class "member name"**
- C6019 (E) The global scope has no class named "class name"**
- C6020 (E) Recursive instantiation of template default argument**
- C6021 (E) Access declarations and using-declarations cannot appear in unions**
- C6022 (E) "name" is not a class member**
- C6028 (W) Invalid redeclaration of nested class**
- C6035 (E) "template name" cannot be declared in this scope**
- C6057 (E) `__evenaccess` qualifier is applied to only integer type**
- C6058 (E) Expected a section name string**
- C6059 (E) Expected a section name**
- C6060 (E) Invalid pragma declaration**
- C6061 (E) "name" has already been specified by other pragma**
- C6062 (E) Pragma may not be specified after definition**
- C6063 (E) Invalid kind of pragma is specified to this symbol**
- C6064 (I) This pragma has no effect**
- C6065 (E) `__regparam?` must be qualified for function type**

**C6066 (E) Illegal attribute specifier**

**C6067 (E) Multiple pointer qualifiers**

**C6068 (E) `_ptr16` must be qualified for data pointer type**

## 12.3 C Library Function Error Messages

For some library functions, if an error is generated during the library function execution, an error number is set in the macro **errno** defined in the header file `<stdio.h>` contained in the standard library. Error messages are defined in the error numbers so that error messages can be output. The following shows an example of an error message output program.

### Example:

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

void main(void)
{
    FILE *fp;

    fp=fopen("file", "w");
    fp=NULL;

    fclose(fp);                                /* error occurred          */

    printf("%s\n", strerror(errno));           /* print error message     */
}
```

### Description:

1. Since the file pointer of NULL is passed to the **fclose** function as an actual parameter, an error will occur. In this case, an error number corresponding to **errno** is set.
2. The **strerror** function returns a pointer of the string literal of the corresponding error message when the error number is passed as an actual parameter. An error message is output by specifying the output of the string literal of the **printf** function.

**Table 12.1 List of C Library Function Error Messages**

Error No.	Error Message/Explanation	Functions to Set Error Numbers
1100 (ERANGE)	DATA OUT OF RANGE An overflow occurred.	frexp, ldexp, modf, ceil, floor, fmod, strtol, atoi, atol, perror, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceilf, cos, cosf, cosh, coshf, exp, expf, floorf, fmodf, ldexpf, log, log10, log10f, logf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexpf
1101 (EDOM)	DATA OUT OF DOMAIN Results for mathematical parameters are not defined.	acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceil, ceilf, cos, cosf, cosh, coshf, exp, expf, floor, floorf, fmod, fmodf, ldexp, ldexpf, log, log10, log10f, logf, modf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexp, frexpf
1102 (EDIV)	DIVISION BY ZERO Division by zero was performed.	div, ldiv
1104 (ESTRN)	TOO LONG STRING The length of string literal exceeds 32767 characters.	strtol, strtod, atof, atoi, atol
1106 (PTRERR)	INVALID FILE POINTER The NULL pointer constant is specified as the file pointer value	fclose, fflush, freopen, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, ungetc, fread, fwrite, fseek, ftell, rewind, perror
1200 (ECBASE)	INVALID RADIX An invalid radix was specified.	strtol, atoi, atol
1202 (ETLN)	NUMBER TOO LONG The specified number exceeds 17 digits.	strtod, fscanf, scanf, sscanf, atof
1204 (EEXP)	EXPONENT TOO LARGE The specified exponent exceeds three digits.	strtod, fscanf, scanf, sscanf, atof
1206 (EEXPN)	NORMALIZED EXPONENT TOO LARGE The exponent exceeds three digits when the string literal is normalized to the IEEE standard decimal format.	strtod, fscanf, scanf, sscanf, atof

**Table 12.1 List of C Library Function Error Messages (cont)**

Error No.	Error Message/Explanation	Functions to Set Error Numbers
1210 (EFLOATO)	OVERFLOW OUT OF FLOAT A float-type decimal value is out of range (overflow).	strtod, fscanf, scanf, sscanf, atof
1220 (EFLOATU)	UNDERFLOW OUT OF FLOAT A float-type decimal value is out of range (underflow).	strtod, fscanf, scanf, sscanf, atof
1230 (EOVER)	FLOATING POINT OVERFLOW A numerical constant exceeds the double type range (overflow).	strtod, fscanf, scanf, sscanf, atof
1240 (EUNDER)	FLOATING POINT UNDERFLOW A numerical constant exceeds the double type range (underflow).	strtod, fscanf, scanf, sscanf, atof
1300 (NOTOPN)	FILE NOT OPEN The file is not open.	fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, vfprintf, vprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, fread, fwrite, fseek, ftell, rewind, perror, freopen
1302 (EBADF)	BAD FILE NUMBER An output function was issued for an input-only file, or an input function was issued for an output-only file.	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, perror, fread, fwrite
1304 (ECSPEC)	ERROR IN FORMAT An erroneous format was specified for an input/output function using format.	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, perror



# Section 13 Assembler Error Messages

## 13.1 Error Message Format and Error Levels

This section gives lists of error messages in order of error code. A list of error messages are provided for each level of errors in the format below:

### **Error code (Error Level: W, E, or F) Error Message**

Meaning of the error message.

Error levels are classified into the following three types:

- (W): Warning error (Continues compiling processing and outputs the object program.)
- (E): Error (Continues compiling processing but does not output the object program.)
- (F): Fatal error (Aborts compiling processing.)

## 13.2 Error Messages

### **10 (E) NO INPUT FILE SPECIFIED**

There is no input source file specified.

Specify an input source file.

### **20 (E) CANNOT OPEN FILE <file name>**

The specified file cannot be opened.

Check and correct the file name and directory.

### **30 (E) INVALID COMMAND PARAMETER**

The options are not correct.

Check and correct the options.

### **40 (E) CANNOT ALLOCATE MEMORY**

All available memory is used up during processing.

This error only occurs when the amount of available user memory is extremely small. If there is other processing occurring at the same time as assembly, interrupt that processing and restart the assembler. If the error still occurs, check and correct the memory management employed on the host computer.

**50 (E) INVALID FILE NAME <file name>**

The file name including the directory is too long or invalid file name.

Check and correct the file name.

It is possible that the object module output by the assembler after this error has occurred will not be usable with the debugger.

**60 (W) INVALID VALUE <file name>**

A value other than 0 is specified for the lower 8 bits of the constant value of the SBR option.

Check the constant value.

The assembler changes the lower 8 bits of the constant value to 0.

**101 (E) SYNTAX ERROR IN SOURCE STATEMENT**

Syntax error in source statement.

Check and correct the whole source statement.

**102 (E) SYNTAX ERROR IN DIRECTIVE**

Syntax error in assembler directive source statement.

Check and correct the whole source statement.

**103 (E) .END NOT FOUND**

.END was not found in the program.

Insert .END in the program.

**104 (E) LOCATION COUNTER OVERFLOW**

The value of location counter exceeded its maximum value.

Reduce the size of the program.

**105 (E) ILLEGAL INSTRUCTION IN STACK SECTION**

An executable instruction or assembler directive that reserves data is in the stack section.

Remove, from the stack section, the executable instruction or assembler directive that reserves data.

**106 (E) TOO MANY ERRORS**

Error display terminated due to too many errors.

Check and correct the whole source statement.

**108 (E) ILLEGAL CONTINUATION LINE**

Illegal continuation line.

Check and correct continuation line.

### **150 (E) INVALID DELAY SLOT INSTRUCTION**

The current delay slot instruction, which is an instruction immediately after a delayed branch instruction, is not allowed.

Check and correct the delay slot instruction by reordering instructions or by another way.

### **200 (E) UNDEFINED SYMBOL REFERENCE**

Undefined symbol reference.

Define the symbol.

### **201 (E) ILLEGAL SYMBOL OR SECTION NAME**

Reserved word (register name, operator, or location counter) specified as symbol or section name.

Correct the symbol or section name.

### **202 (E) ILLEGAL SYMBOL OR SECTION NAME**

Illegal symbol or section name.

Correct the symbol or section name.

### **203 (E) ILLEGAL LOCAL LABEL**

Illegal local label.

Correct the local label.

### **300 (E) ILLEGAL MNEMONIC**

Illegal operation.

Correct the operation.

### **301 (E) TOO MANY OPERANDS OR ILLEGAL COMMENT**

Too many operands of executable instruction, or illegal comment format.

Correct the operands and comment.

### **304 (E) LACKING OPERANDS**

Too few operands.

Correct the operands.

### **306 (E) SYNTAX ERROR IN REGISTER LIST**

Illegal syntax in the register list.

Correct the register list.

### **307 (E) ILLEGAL ADDRESSING MODE OR OBJECT CODE SIZE**

Illegal addressing mode in operand, or illegal allocation size (:8, :16, :24, or :32).

Correct the operand or the allocation size.

**308 (E) SYNTAX ERROR IN OPERAND**

Syntax error in operand.

Correct the operand.

**400 (E) CHARACTER CONSTANT TOO LONG**

Character constant is longer than 4 characters.

Correct the character constant.

**402 (E) ILLEGAL VALUE IN OPERAND**

Operand value out of range for this instruction.

Change the value.

**403 (E) ILLEGAL OPERATION FOR RELATIVE VALUE**

Multiplication, division, or logic operation is specified for a relative-address value.

Correct the expression.

**404 (E) ILLEGAL IMMEDIATE DATA**

A relative value is specified as the operand for #1, #2, #4, #0 to #3, or #0 to #7.

Correct the value.

**407 (E) MEMORY OVERFLOW**

Memory overflow during expression calculation.

Simplify the expression.

**408 (E) DIVISION BY ZERO**

Division by 0 is specified.

Correct the expression.

**409 (E) REGISTER IN EXPRESSION**

Register name in expression.

Correct the expression.

**411 (E) INVALID STARTOF/SIZEOF OPERAND**

STARTOF or SIZEOF specifies illegal section name.

Correct the section name.

**412 (E) ILLEGAL SYMBOL IN EXPRESSION**

Relative-address value or relative symbol is specified as shift value.

Correct the expression.

**413 (E) ILLEGAL DISPLACEMENT VALUE**

The displacement value is illegal.

Make the displacement value even.

**500 (E) SYMBOL NOT FOUND**

Label not defined in directive that requires label.

Insert a label.

**501 (E) ILLEGAL ADDRESS VALUE IN OPERAND**

Illegal specification of the start address or the value of location counter in section.

Correct the start address or value of location counter.

**502 (E) ILLEGAL SYMBOL IN OPERAND**

Illegal value (forward reference symbol, import symbol, relative-address symbol, or undefined symbol) specified in operand.

Correct the operand.

**503 (E) UNDEFINED EXPORT SYMBOL**

Symbol declared for export symbol not defined in the file.

Define the symbol. Alternatively, remove the export symbol declaration.

**504 (E) INVALID RELATIVE SYMBOL IN OPERAND**

Illegal value (forward reference symbol or import symbol) specified in operand.

Correct the operand.

**505 (E) ILLEGAL OPERAND**

Misspelled operand.

Correct the operand.

**506 (E) ILLEGAL OPERAND**

Illegal element specified in operand.

Correct the operand.

**508 (E) ILLEGAL VALUE IN OPERAND**

Operand value out of range for this directive.

Correct the operand.

**510 (E) ILLEGAL BOUNDARY VALUE**

Illegal boundary alignment value.

Correct the boundary alignment value.

**511 (E) ILLEGAL DISPLACEMENT SIZE**

Illegal number of bits for .DISPSIZE.

Correct the number of bits.

### **512 (E) ILLEGAL EXECUTION START ADDRESS**

Illegal execution start address.

Correct the execution start address.

### **513 (E) ILLEGAL REGISTER NAME**

Illegal register name.

Correct the register name.

### **514 (E) INVALID EXPORT SYMBOL**

Symbol declared for export symbol that cannot be exported.

Remove the declaration for the export symbol.

### **516 (E) EXCLUSIVE DIRECTIVES**

Inconsistent directive specification.

Check and correct all related directives.

### **517 (E) INVALID VALUE IN OPERAND**

Illegal value (forward reference symbol, import symbol, or relative-address symbol in other sections) specified in operand.

Correct the operand.

### **518 (E) INVALID IMPORT SYMBOL**

Symbol declared for import symbol defined in the file.

Remove the declaration for the import symbol.

### **520 (E) ILLEGAL .CPU DIRECTIVE POSITION**

.CPU is not specified at the beginning of the program, or specified more than once.

Specify .CPU at the beginning of the program once.

### **521 (E) ILLEGAL SYMBOL IN OPERAND**

In the **optimize** option specification, a symbol that has an address as a value or a location counter value is specified for the operand that requires a constant value.

Do not use the **optimize** option when specifying a symbol that has an address as a value or a location counter value.

### **523 (E) ILLEGAL OPERAND**

Illegal .LINE directive operand.

Correct the operand.

### **524 (E) ILLEGAL ADDRESSING SPACE SIZE**

Illegal address-area bit width is specified as the operand of the .CPU directive.

Correct the address-area bit width.

#### **525 (E) ILLEGAL .LINE DIRECTIVE POSITION**

.LINE directive specified during macro expansion or conditional iterated expansion.  
Change the specified position of the .LINE directive.

#### **526 (E) STRING TOO LONG**

The operand string literal has more than 255 characters.

The string literals to specify to the operand of .SDATA, .SDATAB, SDATAC, and SDATAZ directives must have 255 or less characters.

#### **527 (E) CANNOT SUPPORT COMMON SECTION SINCE VERSION 4**

COMMON is specified for the section attribute.

Common section cannot be used.

More than one section can be allocated to the same address by using a colon (:) in the **start** option of the optimizing linkage editor.

#### **528 (E) SPECIFICATION OF THE ADDRESS OVERLAPS**

Address allocation overlaps in a section.

Check the specified contents of .SECTION and .ORG directive.

#### **529 (E) THE ADDRESS BETWEEN SECTIONS OVERLAPS**

Address allocation overlaps between sections.

Check the specified contents of .SECTION and .ORG directive.

#### **532 (E) ILLEGAL OPERAND**

Error in the operand of .STACK.

Correct the stack value to be multiples of 2.

#### **533 (E) ILLEGAL .STACK DIRECTIVE POSITION**

.STACK is specified in macro expansion or conditional iterated expansion.

Correct the location of .STACK.

#### **600 (E) INVALID CHARACTER**

Illegal character.

Correct it.

#### **601 (E) INVALID DELIMITER**

Illegal delimiter character.

Correct it.

#### **602 (E) INVALID CHARACTER STRING FORMAT**

String literal error.

Correct it.

### **603 (E) SYNTAX ERROR IN SOURCE STATEMENT**

Source statement syntax error.

Reexamine the whole source statement.

### **604 (E) ILLEGAL SYMBOL IN OPERAND**

Illegal operand specified in a directive.

No symbol or location counter (\$) can be specified as an operand of this directive.

### **610 (E) MULTIPLE MACRO NAMES**

Macro name reused in macro definition (.MACRO directive).

Correct the macro name.

### **611 (E) MACRO NAME NOT FOUND**

Macro name not specified (.MACRO directive).

Specify a macro name.

### **612 (E) ILLEGAL MACRO NAME**

Macro name error (.MACRO directive).

A macro name cannot be a mnemonic of an executable instruction, directive (excluding a period (.)), or directive statement (excluding a period (.)).

Correct the macro name.

### **613 (E) ILLEGAL .MACRO DIRECTIVE POSITION**

.MACRO directive appears in macro body (between .MACRO and .ENDM directives), between .AREPEAT and .AENDR directives, or between .AWHILE and .AENDW directives. Remove the .MACRO directive.

### **614 (E) MULTIPLE MACRO PARAMETERS**

Identical arguments repeated in argument declaration in macro definition (.MACRO directive).

Correct the arguments.

### **615 (E) ILLEGAL .END DIRECTIVE POSITION**

.END directive appears in macro body (between .MACRO and .ENDM directives).

Remove the .END directive.

### **616 (E) MACRO DIRECTIVES MISMATCH**

An .ENDM directive appears without a preceding .MACRO directive, or an .EXITM directive appears outside of a macro body (between .MACRO and .ENDM directives), outside of .AREPEAT and .AENDR directives, or outside of .AWHILE and .AENDW directives.

Remove the .ENDM or .EXITM directive.



**618 (E) MACRO EXPANSION TOO LONG**

Line with over 8,192 characters generated by macro expansion.

Correct the definition or call so that the line is less than or equal to 8,192 characters.

**619 (E) ILLEGAL MACRO PARAMETER**

Macro parameter name error in macro call, or error in argument in a macro body (between .MACRO and .ENDM directives).

Correct the argument.

When there is an error in a argument in a macro body, the error will be detected and flagged during macro expansion.

**620 (E) UNDEFINED PREPROCESSOR VARIABLE**

Reference to an undefined preprocessor variable.

Define the preprocessor variable.

**621 (E) ILLEGAL .END DIRECTIVE POSITION**

.END directive in macro expansion.

Remove the .END directive.

**622 (E) ')' NOT FOUND**

Matching parenthesis missing in macro processing exclusion.

Add the macro processing exclusion parenthesis.

**623 (E) SYNTAX ERROR IN STRING FUNCTION**

Syntax error in string literal manipulation function.

Correct the string literal manipulation function.

**624 (E) MACRO PARAMETERS MISMATCH**

Too many macro parameters for positional specification in macro call.

Correct the number of macro parameters.

**630 (E) SYNTAX ERROR IN OPERAND**

Syntax error in the operand of the structured assembly directive statement.

Reexamine the whole source statement.

**631 (E) END DIRECTIVE MISMATCH**

Terminating preprocessor directive does not agree with matching directive.

Reexamine the preprocessor directives.

**632 (E) SYNTAX ERROR IN OPERAND**

Syntax error in the operand condition code of a structured assembly directive statement.

Correct the condition code.

**633 (E) ILLEGAL .BREAK OR .CONTINUE DIRECTIVE POSITION**

.BREAK or .CONTINUE is outside the .FOR[U] and .ENDF, .WHILE and .ENDW, or .REPEAT and .UNTIL.

Remove .BREAK or .CONTINUE.

**634 (E) EXPANSION TOO LONG**

The number of characters in one line of a structured assembly expansion exceeds 8,192 characters. Correct the program so that the number of characters in one line is 8,192 or less.

**640 (E) SYNTAX ERROR IN OPERAND**

Syntax error in conditional assembly directive statement operand.

Reexamine the entire source statement.

**641 (E) INVALID RELATIONAL OPERATOR**

Error in conditional assembly directive statement relational operator.

Correct the relational operator.

**642 (E) ILLEGAL .END DIRECTIVE POSITION**

.END directive appears between .AREPEAT and .AENDR directives or between .AWHILE and .AENDW directives.

Remove the .END directive.

**643 (E) DIRECTIVE MISMATCH**

.AENDR or .AENDW directive does not form a proper pair with .AREPEAT or .AWHILE directive.

Reexamine the preprocessor directives.

**644 (E) ILLEGAL .AENDW OR .AENDR DIRECTIVE POSITION**

.AENDW or .AENDR directive appears between .AIF and .AENDI directives.

Remove the .AENDW or .AENDR directive.

**645 (E) EXPANSION TOO LONG**

Line with over 8,192 characters generated by .AREPEAT or .AWHILE expansion.

Correct the .AREPEAT or .AWHILE to generate lines of less than or equal to 8,192 characters.

**650 (E) INVALID INCLUDE FILE**

Error in .INCLUDE file name.

Correct the file name.

**651 (E) CANNOT OPEN INCLUDE FILE**

Cannot open .INCLUDE file name.

Correct the file name.

**652 (E) INCLUDE NEST TOO DEEP**

File inclusion nesting exceeded 30 levels.

Limit the nesting to 30 or fewer levels.

**653 (E) SYNTAX ERROR IN OPERAND**

Syntax error in .INCLUDE operand.

Correct the operand.

**660 (E) .ENDM NOT FOUND**

Missing .ENDM directive following .MACRO.

Insert an .ENDM directive.

**661 (E) .END DIRECTIVE NOT FOUND**

A .END directive was not found in the structured assembly directive statement.

Insert a .END directive.

**662 (E) ILLEGAL .END DIRECTIVE POSITION**

.END directive appears between .AIF and .AENDI.

Remove the .END directive.

**663 (E) ILLEGAL .END DIRECTIVE POSITION**

.END directive appears in included file.

Remove the .END directive.

**664 (E) ILLEGAL .END DIRECTIVE POSITION**

.END directive appears between .AIF and .AENDI directives.

Remove the .END directive.

**665 (E) ILLEGAL SYMBOL IN OPERAND**

A symbol other than the preprocessor variable is specified for the preprocessor directive in the **optimize** option specification. Correct the symbol.

Do not use the **optimize** option when specifying a symbol other than the preprocessor variable.

**667 (E) EXPANSION TOO LONG**

Lines with over 8,192 characters are generated by the .DEFINE directive.

Correct the .DEFINE directive to generate lines of less than or equal to 8,192 characters.

**668 (E) ILLEGAL VALUE IN OPERAND**

Error in the operand of the .AIFDEF directive.

Specify, as the operand of this directive, a symbol defined by .DEFINE directive.

### **669 (E) STRING TOO LONG**

The operand string literal has more than 255 characters.

The string literals to specify to the operand of .ASSIGNC directive, .DEFINE directive, and character manipulating functions (.LEN, .INSTR, .SUBSTR) must have 255 or less characters.

### **670 (E) SUCCESSFUL CONDITION .AERROR**

A statement including the .AERROR directive has been processed by the condition of .AIF.

Check the condition statement to avoid .AERROR processing.

### **800 (W) SYMBOL NAME TOO LONG**

A preprocessor variable or the **define** replacement symbol exceeded 33 characters.

Correct the symbol.

The assembler ignores the characters starting at the 33rd character.

### **801 (W) MULTIPLE SYMBOLS**

Symbol already defined.

Remove the symbol redefinition.

The assembler ignores the second and later definitions.

### **805 (W) ILLEGAL OPERATION SIZE**

An illegal branch size (:8 or :16) was set for a structured assembly directive statement.

Correct the branch size.

### **807 (W) ILLEGAL OPERATION SIZE**

Illegal operation size.

Correct the operation size.

The assembler ignores the incorrect operation size specification.

### **808 (W) ILLEGAL CONSTANT SIZE**

Illegal notation for an integer constant.

Correct the notation.

The size is either byte (.B) or word (.W), which is signed one and two byte values, respectively.

### **810 (W) TOO MANY OPERANDS**

Too many operands or illegal comment format.

Correct the operand or the comment.

The assembler ignores the extra operands.

### **811 (W) ILLEGAL SYMBOL DEFINITION**

A label specified in assembler directive that cannot have a label is written.

Remove the label specification.

The assembler ignores the label.

### **813 (W) SECTION ATTRIBUTE MISMATCH**

A different section type is specified on section restart (reentry), or a section start address is respecified at the restart of absolute-address section.

Do not respecify the section type or start address on section reentry.

The specification of starting section remains valid.

### **814 (W) ILLEGAL OBJECT CODE SIZE**

Illegal allocation size (:8, :16, :24, or :32).

Correct the size.

#xx:2 and #xx:3 are symbols used in the manual, and cannot be used in the actual assembly language.

### **815 (W) MULTIPLE MODULE NAMES**

Respecification of object module name.

Specify the object module name once in a program.

The assembler ignores the second and later object module name specifications.

### **816 (W) START ODD ADDRESS**

An even number of bytes or area of data start at an odd address.

Correct the address to an even address.

### **817 (W) OPERATION SIZE MISMATCH**

@-SP or @SP+ is specified for a byte-sized (.B) operand.

Object code is still output, but this specification should be avoided since the SP (stack pointer) will then have an odd value.

### **818 (W) ILLEGAL ACCESS SIZE**

Illegal access size (:8 or :16).

Correct the access size.

### **819 (W) @Rn+, @-Rn, @+Rn, @Rn-, @(d,Rn) OR @Rn USED**

Use ERn instead of Rn in @Rn+, @-Rn, @+Rn, @Rn-, @(d,Rn), or @Rn with the H8/300H, H8S or H8SX CPU.

### **825 (W) ILLEGAL INSTRUCTION IN DUMMY SECTION**

An executable instruction or assembler directive that reserves data is in dummy section.

Remove, from the dummy section, the executable instruction or assembler directive that reserves data.

The assembler ignores the executable instruction or assembler directive that reserves data in dummy section.

### **830 (W) OPERATION SIZE MISMATCH**

ERn or Rn is specified for a byte-sized (.B) operand, or ERn is specified for a word-sized (.W) operand.

Correct the register specification.

Object code is generated assuming RnL for byte size operand and Rn for word size operand.

### **832 (W) MULTIPLE 'P' DEFINITIONS**

Symbol P already defined when a default section is used.

Do not define P as a symbol if a default section is used.

The assembler regards P as the name of the default section, and ignores other definitions of the symbol P.

### **835 (W) ILLEGAL VALUE IN OPERAND**

Operand value out of range for an executable instruction.

Correct the value.

The assembler generates object code with a value corrected to be within range.

### **836 (W) CONSTANT SIZE OVERFLOW**

An integer constant value is outside the range of possible sizes (.B or .W).

Correct the integer constant value.

The assembler interprets the size as a byte (.B) or word (.W), 1- or 2-byte signed values, respectively.

### **837 (W) SOURCE STATEMENT TOO LONG**

The length of a source statement exceeded 8,192 bytes.

Rewrite the source statement to be within 8,192 bytes by, for example, rewriting the comment.

Alternatively, rewrite the statement as a multi-line statement.

### **838 (W) ILLEGAL CHARACTER CODE**

The shift JIS code, EUC code, or LATIN1 code is specified outside string literals and comments, or the **sjis**, **euc**, or **latin1** option is not specified.

Specify the shift JIS code or EUC code in string literals or comments, or specify the **sjis**, **euc**, or **latin1** option.

### **850 (W) ILLEGAL SYMBOL DEFINITION**

Symbol specified in label field.

Remove the symbol.

### **851 (W) MACRO SERIAL NUMBER OVERFLOW**

Macro generation counter exceeded 99,999.

Reduce the number of macro calls.

## **852 (W) UNNECESSARY CHARACTER**

Characters appear after the operands.

Correct the operand(s).

## **853 (W) NEGATIVE IMMEDIATE VALUE**

#-xx is specified for the increased value of .FOR[U].

Correct #-xx to -#xx.

The assembler will expand .FOR[U] as is.

## **854 (W) .AWHILE ABORTED BY .ALIMIT**

Expansion count has reached the maximum value specified by .ALIMIT directive, and expansion has been terminated.

Check the condition for iterated expansion.

## **855 (W) ILLEGAL VALUE IN OPERAND**

A value other than 0 is specified for the lower 8 bits of the constant value of the SBR directive.

Check the constant value.

The assembler changes the lower 8 bits of the constant value to 0.

## **856 (W) MULTIPLE SYMBOLS**

A stack value is defined for the same symbol again.

Remove the stack value redefinition.

The assembler ignores the second and later definitions.

## **870 (W) ILLEGAL DISPLACEMENT VALUE**

The displacement value is illegal.

Make the displacement value even.

The assembler generates the object code as it was written.

## **871 (W) MISSING DELAY SLOT INSTRUCTION**

The delay slot instruction, which would be an instruction immediately after a delayed branch instruction, is missing.

Check and add the delay slot instruction by reordering instructions or by another way.

The assembler generates the object code as it was written.

## **901 (F) SOURCE FILE INPUT ERROR**

Source file input error.

Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.

## **902 (F) MEMORY OVERFLOW**

Insufficient memory. (Unable to process the temporary information.)

Subdivide the program.

### **903 (F) LISTING FILE OUTPUT ERROR**

Output error on the list file.

Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.

### **904 (F) OBJECT FILE OUTPUT ERROR**

Output error on the object file.

Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.

### **905 (F) MEMORY OVERFLOW**

Insufficient memory. (Unable to process the line information.)

Subdivide the program.

### **906 (F) MEMORY OVERFLOW**

Insufficient memory. (Unable to process the symbol information.)

Subdivide the program.

### **907 (F) MEMORY OVERFLOW**

Insufficient memory. (Unable to process the section information.)

Subdivide the program.

### **908 (F) SECTION OVERFLOW**

Too much number of sections.

When debugging information is output, up to 32,633 sections is enabled.

When debugging information is not output, up to 32,638 sections is enabled.

Subdivide the program.

### **933 (F) LACKING CPU SPECIFICATION**

The CPU type was not specified.

Specify the CPU type using the **cpu** option, a .CPU directive, or the H38CPU environment variable.

### **935 (F) SUBCOMMAND FILE INPUT ERROR**

Subcommand file input error.

Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.

### **954 (F) MEMORY OVERFLOW**

Insufficient memory.

Subdivide the source program.



#### **955 (F) LOCAL BLOCK NUMBER OVERFLOW**

The number of local blocks that are valid in the local label exceeded 100,000.  
Subdivide the source program.

#### **956 (F) EXPAND FILE INPUT/OUTPUT ERROR**

File output error for preprocessor expansion.  
Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.

#### **957 (F) MEMORY OVERFLOW**

Insufficient memory.  
Subdivide the source program.

#### **964 (F) MEMORY OVERFLOW**

Insufficient memory.  
Information on symbols cannot be processed.  
Subdivide the source program.

#### **970 (F) MEMORY OVERFLOW**

Insufficient memory.  
Section size is too large. A large offset may have been given to the location counter using a .ORG directive, or a large data area may have been reserved by using directives such as .DATAB.  
Subdivide the section or reduce the data area.



# Section 14 Error Messages for the Optimizing Linkage Editor

## 14.1 Error Format and Error Levels

In this section, error messages output in the following format and the details of errors are explained.

Error number            (Error level) Error message

Error details

There are five different error levels, corresponding to different degrees of seriousness.

Error Number	Error Level	Error Type	Description
L0000–L0999 P0000–P0999	(I)	Information	Processing is continued.
L1000–L1999 P1000–P1999	(W)	Warning	Processing is continued.
L2000–L2999 P2000–P2999	(E)	Error	Option analysis processing is continued; processing is interrupted.
L3000–L3999 P3000–P3999	(F)	Fatal	Processing is interrupted.
L4000– P4000–	(–)	Internal	Processing is interrupted.

Error numbers beginning with an L are optimizing linkage editor output messages.

Error numbers beginning with a P are prelinker output messages. Output of errors with numbers beginning with a P cannot be controlled using the `nomessage` or `change_message` options.

## 14.2 List of Messages

### L0001 (I) Section “section” created by optimization “optimization”

The section named **section** was created as a result of the **optimization**.

### L0002 (I) Symbol “symbol” created by optimization “optimization”

The symbol named **symbol** was created as a result of the **optimization**.

#### **L0003 (I) “file”-“symbol” moved to “section” by optimization**

As a result of variable\_access optimization, the symbol named **symbol** in **file** was moved.

#### **L0004 (I) “file”-“symbol” deleted by optimization**

As a result of symbol\_delete optimization, the symbol named **symbol** in **file** was deleted.

#### **L0005 (I) The offset value from the symbol location has been changed by optimization : “file”-“section”-“symbol ± offset”**

As a result of the size being changed by optimization within the range of **symbol ± offset**, the offset value was changed. Check that this does not cause a problem. To disable changing of the offset value, cancel the specification of the goptimize option on assembly of **file**.

#### **L0100 (I) No inter-module optimization information in “file”**

No inter-module optimization information was found in **file**. Inter-module optimization is not performed on **file**. To perform inter-module optimization, specify the goptimize option on compiling and assembly. Note however that the goptimize option is not available in asmsh.

#### **L0101 (I) No stack information in “file”**

No stack information was found in **file**. **file** may be an assembler output file or a SYSROF-> ELF converted file. The contents of the file will not be in the stack information file output by the optimizing linkage editor.

#### **L0102 (I) Stack size “size” specified to the undefined symbol “symbol” in “file”**

Stack size **size** is specified for the undefined symbol named **symbol** in **file**

#### **L0103 (I) Multiple stack sizes specified to the symbol “symbol”**

Multiple stack sizes are specified for the symbol named **symbol**.

#### **P0200 (I) “instance” no longer needed in “file”**

An unused instance named **instance** exists in **file**.

#### **P0201 (I) “instance” assigned to file “file”**

The instance named **instance** was assigned to **file**.

#### **P0202 (I) Executing : “command”**

The command **command** is being executed in order to generate an instance.

#### **P0203 (I) “instance” adopted by file “file”**

The instance named **instance** was assigned to **file**.

#### **L0300 (I) Mode type “mode type 1” in “file” differ from “mode type 2”**

A file with a different mode type was input.

**L0400 (I) Unused symbol “file”-“symbol”**

The symbol named **symbol** in **file** is not used.

**L1000 (W) Option “option” ignored**

The option named **option** is invalid, and is ignored.

**L1001 (W) Option “option 1” is ineffective without option “option 2”**

**option 1** needs specifying **option 2**. **option 1** is ignored.

**L1002 (W) Option “option 1” cannot be combined with option “option 2”**

**option 1** and **option 2** cannot be specified simultaneously. **option 1** is ignored.

**L1003 (W) Divided output file cannot be combined with option “option”**

**option** and the option to divide the output file cannot be specified simultaneously. **option** is ignored. The first input file name is used as the output file name.

**L1004 (W) Fatal level message cannot be changed to other level : “number”**

The level of a fatal error type message cannot be changed. The specification of **number** is ignored. Only errors at the information/warning/error level can be changed with the `change_message` option.

**L1005 (W) Subcommand file terminated with end option instead of exit option**

There is no processing specification following the end option. Processing is done with the exit option assumed.

**L1006 (W) Options following exit option ignored**

All options following the exit option is ignored.

**L1007 (W) Duplicate option : “option”**

Duplicate specifications of **option** were found. Only the last specification is effective.

**L1008 (W) Option “option” is effective only in cpu type “CPU type”**

**option** is effective only in **CPU type**. **option** is ignored.

**L1011 (W) Duplicate module specified in option “option” : “module”**

**option** was used to specify the same module twice. The second specification is ignored.

**L1012 (W) Duplicate symbol/section specified in option “option” : “name”**

**option** was used to specify the same symbol name or section name twice. The second specification is ignored.

**L1013 (W) Duplicate number specified in option “option” : “number”**

**option** was used to specify the same error number. Only the last specification is effective.

**L1100 (W) Cannot find “name” specified in option “option”**

The symbol name or section name specified in **option** cannot be found. The **name** specification is ignored.

**L1101 (W) “name” in rename option conflicts between symbol and section**

**name** specified by the rename option exists as both a section name and as a symbol name. Rename is performed for the symbol name only in this case.

**L1102 (W) Symbol “symbol” redefined in option “option”**

The symbol specified by **option** has already been defined. Processing is continued without any change.

**L1103 (W) Invalid address value specified in option “option” : “address”**

**address** specified by **option** is invalid. The **address** specification is ignored.

**L1104 (W) Invalid section specified in option “option” : “section”**

A section without an initial value cannot be specified by **option**. The **section** specification is ignored.

**L1110 (W) Entry symbol “symbol” in entry option conflicts**

A symbol other than **symbol** specified by the entry option is specified as the entry symbol on compiling or assembling. The option specification is given priority.

**L1120 (W) Section address is not assigned to “section”**

There is no specification of the address to **section**. **section** is placed at the end.

**L1121 (W) Address cannot be assigned to absolute section “section” in start option**

**section** is an absolute address section. An address assigned to an absolute address section is ignored.

**L1122 (W) Section address in start option is incompatible with alignment : “section”**

The address of **section** specified by the start option conflicts with memory boundary alignment requirements. The section address is modified to conform to boundary alignment.

**L1130 (W) Section attribute mismatch in rom option : “section 1, section 2”**

The attributes and boundary alignment of **section 1** and **section 2** specified by the rom option are different. The larger value is effective as the boundary alignment of **section 2**.

**L1140 (W) Load address overflowed out of record-type in option “option”**

A record type smaller than the address value was specified. The range exceeding the specified record type has been output as different record type.

#### **L1141 (W) Cannot fill unused area from “address” with the specified value**

Specified data cannot be output to addresses higher than **address** because the unused area size is not a multiple of the value specified by the space option.

#### **L1150 (W) Sections in fsymbol option have no symbol**

Sections specified by the fsymbol option have no externally defined symbols. The fsymbol option has been ignored.

#### **L1160 (W) Undefined external symbol “symbol”**

An undefined external symbol **symbol** was referenced.

#### **L1170 (W) Specified SBR addresses conflict**

Different SBR addresses have been specified. Processing is done with SBR=USER assumed.

#### **L1171 (W) Least significant byte in SBR=“constant” ignored**

The least significant 8 bits in address **constant** specified by the SBR option are ignored.

#### **L1200 (W) Backed up file “file 1” into “file 2”**

The file **file 1** was backed up to the file **file 2**.

#### **L1300 (W) No debug information in input files**

There is no debugging information in the input files. The debug, sdebug, or compress option has been ignored. Check whether the relevant option was specified at compilation or assembly.

#### **L1301 (W) No inter-module optimization information in input files**

No inter-module optimization information is present in the input files. The optimize option has been ignored. Check whether the goptimize option was specified at compilation or assembly.

#### **L1302 (W) No stack information in input files**

No stack information is present in the input files. The stack option is ignored. If all input files are assembler output files or SYSROF->ELF converted files, the stack option is ignored.

#### **L1310 (W) “section” in “file” is not supported in this tool**

An unsupported section was present in **file**. **section** has been ignored.

#### **L1311 (W) Invalid debug information format in “file”**

Debugging information in **file** is not dwarf2. The debugging information has been deleted.

#### **L1320 (W) Duplicate symbol “symbol” in “file”**

The symbol named **symbol** is duplicated. The symbol in the first file input is given priority.

#### **L1321 (W) Entry symbol “symbol” in “file” conflicts**

Multiple object files containing more than one entry symbol definition were input. Only the entry symbol in the first file input is effective.

#### **L1322 (W) Section alignment mismatch : “section”**

Sections with the same name but different boundary alignments were input. Only the largest boundary alignment specification is effective.

#### **L1323 (W) Section attribute mismatch : “section”**

Sections with the same name but different attributes were input. If they are an absolute section and relative section, the section is treated as an absolute section. If the read/write attributes mismatch, both are allowed.

#### **L1324 (W) Symbol size mismatch : “symbol” in “file”**

Common symbols or defined symbols with different sizes were input. A defined symbol is given priority. In the case of two common symbols, the symbol in the first file input is given priority.

#### **L1330 (W) Cpu type “CPU type 1” in “file” differ from “CPU type 2”**

Files with different CPU types were input. Processing is continued with the CPU type assumed as H8SX.

#### **L1400 (W) Stack size overflow in register optimization**

During register optimization, the stack access code exceeded the stack size limit of the compiler. The register optimization specification has been ignored.

#### **L1401 (W) Function call nest too deep**

The number of function call nesting levels is so deep that register optimization cannot be performed.

#### **L1410 (W) Cannot optimize “file”-“section” due to multi label relocation operation**

A section having multiple label relocation operations cannot be optimized. Section **section** in file **file** has not been optimized.

#### **L1420 (W) “file” is newer than “profile”**

**file** was updated after **profile**. The profile information has been ignored.

#### **L1500 (W) Cannot check stack size**

There is no stack section, and so consistency of the stack size specified by the stack option on compiling cannot be checked. To check the consistency of the stack size on compiling, the optimize option needs to be specified on compiling and assembling.

#### **L1501 (W) Stack size overflow : “stack size”**

The stack section size exceeded the **stack size** specified by the stack option on compiling. Either change the option used on compiling, or change the program so as to reduce the use of the stack.



**L1502 (W) Stack size in “file” conflicts with that in another file**

Different values for stack size are specified for multiple files. Check the options used on compiling.

**P1600 (W) An error occurred during name decoding of “instance”**

**instance** could not be decoded. The message is output using the encoding name.

**L2000 (E) Invalid option : “option”**

**option** is not supported.

**L2001 (E) Option “option” cannot be specified on command line**

**option** cannot be specified on the command line. Specify this option in a subcommand file.

**L2002 (E) Input option cannot be specified on command line**

The input option was specified on the command line. Input file specification on the command line should be made without the input option.

**L2003 (E) Subcommand option cannot be specified in subcommand file**

The subcommand option was specified in a subcommand file. The subcommand option cannot be nested.

**L2004 (E) Option “option 1” cannot be combined with option “option 2”**

**option 1** and **option 2** cannot be specified simultaneously.

**L2005 (E) Option “option” cannot be specified while processing “process”**

**option** cannot be specified to **process**.

**L2006 (E) Option “option 1” is ineffective without option “option 2”**

**option 1** requires **option 2** be specified.

**L2010 (E) Option “option” requires parameter**

**option** requires a parameter to be specified.

**L2011 (E) Invalid parameter specified in option “option” : “parameter”**

An invalid parameter was specified for **option**.

**L2012 (E) Invalid number specified in option “option” : “value”**

An invalid value was specified for **option**. Check the range of valid values.

**L2013 (E) Invalid address value specified in option “option” : “address”**

The address **address** specified in **option** is invalid. A hexadecimal address between 0 and FFFFFFFF should be specified.

**L2014 (E) Illegal symbol/section name specified in “option” : “name”**

The section or symbol name specified in **option** uses an illegal character. Only alphanumeric, the underscore (\_), and the dollar sign (\$) may be used in section/symbol names (the leading character cannot be a number).

**L2016 (E) Invalid alignment value specified in option “option” : “alignment value”**

The **alignment value** specified in **option** is invalid. A power of 2(1, 2, 4, 8, 16, or 32) should be specified in decimal.

**L2020 (E) Duplicate file specified in option “option” : “file”**

The same file was specified twice in **option**.

**L2021 (E) Duplicate symbol/section specified in option “option” : “name”**

The same symbol name or section name was specified twice in **option**.

**L2022 (E) Address ranges overlap in option “option” : “address range”**

Address ranges **address range** specified in **option** overlap.

**L2100 (E) Invalid address specified in cpu option : “address”**

An invalid address was specified in the cpu option.

**L2101 (E) Invalid address specified in option “option” : “address”**

The address specified in **option** exceeds the address range that can be specified by the **cpu** or the range specified by the **cpu** option.

**L2110 (E) Section size of second parameter in rom option is not 0 : “section”**

A section whose size is not zero was specified in the second parameter of the rom option.

**L2111 (E) Absolute section cannot be specified in rom option : “section”**

An absolute address section was specified in the rom option.

**L2120 (E) Library “file” without module name specified as input file**

A library file without a module name was specified as the input file.

**L2121 (E) Input file is not library file : “file (module)”**

The file specified by **file (module)** as the input file is not a library file.

**L2130 (E) Cannot find file specified in option “option” : “file”**

The file specified in **option** could not be found.

**L2131 (E) Cannot find module specified in option “option” : “module”**

The module specified in **option** could not be found.

**L2132 (E) Cannot find “name” specified in option “option”**

The symbol or section specified in **option** does not exist.

**L2133 (E) Cannot find defined symbol “name” in option “option”**

The externally defined symbol specified in **option** does not exist.

**L2140 (E) Symbol/section “name” redefined in option “option”**

The symbol or section specified in **option** has already been defined.

**L2141 (E) Module “module” redefined in option “option”**

The module specified in **option** has already been defined.

**L2200 (E) Illegal object file : “file”**

**P2200**

A format other than ELF format was input.

**L2201 (E) Illegal library file : “file”**

**file** is not a library file.

**L2202 (E) Illegal cpu information file : “file”**

**file** is not a cpu information file.

**L2203 (E) Illegal profile information file : “file”**

**file** is not a profile information file.

**L2210 (E) Invalid input file type specified for option “option” : “file (type)”**

When specifying **option**, a file (type) that cannot be processed was input.

**L2211 (E) Invalid input file type specified while processing “process” : “file (type)”**

A file (type) that cannot be processed was input during processing **process**.

**L2220 (E) Illegal mode type “mode type” in “file”**

A file with a different mode type was input.

**L2221 (E) Section type mismatch : “section”**

Sections with the same name but different attributes (whether initial values present or not) were input.

**L2300 (E) Duplicate symbol “symbol” in “file”**

There are duplicate occurrences of **symbol**.

**L2301 (E) Duplicate module “module” in “file”**

There are duplicate occurrences of **module**.

### **L2310 (E) Undefined external symbol “symbol” referenced in “file”**

An undefined symbol **symbol** was referenced in **file**.

### **L2311 (E) Section “section 1” cannot refer to overlaid section : “section 2”-“symbol”**

A symbol defined in **section 1** was referenced in **section 2** that is allocated to the same address as **section 1** overlaid. **section 1** and **section 2** must not be allocated to the same address.

### **L2320 (E) Section address overflowed out of range : “section”**

The address of **section** exceeds the usable address range.

### **L2321 (E) Section “section 1” overlaps section “section 2”**

The addresses of **section 1** and **section 2** overlap. Change the address specified by the start option.

### **L2322 (E) Section size too large: “section”**

The size of section is too large. The size of a \$TBR section must be 1024 bytes or less.

### **L2330 (E) Relocation size overflow : “file”-“section”-“offset”**

The result of the relocation operation exceeded the relocation size. Possible causes include inaccessibility of a branch destination, and referencing of a symbol which must be located at a specific address. Ensure that the referenced symbol at the **offset** position of **section** in the compile or assembly list is placed at the correct position.

### **L2331 (E) Division by zero in relocation value calculation : “file”-“section”-“offset”**

Division by zero occurred during a relocation operation. Check for problems in calculation of the position at **offset** in **section** in the compile or assembly list.

### **L2332 (E) Relocation value is odd number : “file”-“section”-“offset”**

The result of the relocation operation is an odd number. Check for problems in calculation of the position at **offset** in **section** in the compile or assembly list.

### **L2340 (E) Symbol name in section “section” is too long**

The number of characters in symbols in **section** specified by fsymbol exceeded 8174.

### **L2400 (E) Global register in “file” conflicts : “symbol”, “register”**

Another symbol has already been allocated to a global register specified in **file**.

### **L2401 (E) \_\_near8, \_\_near16 symbol “symbol” is outside near memory area**

**symbol** is not allocated in the **\_\_near8** or **\_\_near16** range. Either change the start specification, or remove the **\_\_near** specifier at compilation, so that correct address calculations can be made.

### **L2402 (E) Number of register parameter conflicts with that in another file : “function”**

Different numbers of register parameters are specified for **function** in multiple files.

**L2410 (E) Address value specified by map file differs from one after linkage as to “symbol”**

The address of **symbol** is different between the address within the external symbol allocation information file used at compilation and the address after linkage.

Check whether the program has not been changed before and after specification of the map option at compilation. optlnk optimization may cause the sequence of the symbols to differ before and after specification of the map option at compilation. Disable the map option at compilation or disable the optlnk option for optimization.

**L2411 (E) Map file in “file” conflicts with that in another file**

Different external symbol allocation information files were used by the input files at compilation.

**L2412 (E) Cannot open file : “file”**

**file** (external symbol allocation information file) cannot be opened. Check whether the file name and access rights are correct.

**L2413 (E) Cannot close file : “file”**

**file** (external symbol allocation information file) cannot be closed. There may be insufficient disk space.

**L2414 (E) Cannot read file : “file”**

**file** (external symbol allocation information file) cannot be read. An empty file may have been input, or there may be insufficient disk space.

**L2415 (E) Illegal map file : “file”**

**file** (external symbol allocation information file) has an illegal format. Check whether the file name is correct.

**L2416 (E) Order of functions specified by map file differs from one after linkage as to “function name”**

The sequential position of the function "function name" in the functions differs between the position in the information of the external symbol allocation information file used at compilation and the position after linkage. The address of a **static** variable within the function may differ between the external symbol allocation information file and the result after linkage.

**P2500 (E) Cannot find library file : “file”**

**file** specified as a library file cannot be found.

**P2501 (E) “instance” has been referenced as both an explicit specialization and a generated instantiation**

Instantiation has been requested of an instance already defined. For the file using **instance**, confirm that **form=relocate** has not been used to generate a relocatable object file.

### **P2502 (E) “instance” assigned to “file 1” and “file 2”**

The definition of **instance** is duplicated in **file 1** and **file 2**. For the file using **instance**, confirm that **form=relocate** has not been used to generate a relocatable object file.

### **L3000 (F) No input file**

There is no input file.

### **L3001 (F) No module in library**

There are no modules in the library.

### **L3002 (F) Option “option 1” is ineffective without option “option 2”**

The option **option 1** requires that the option **option 2** be specified.

### **L3100 (F) Section address overflow out of range : “section”**

The address of **section** exceeded FFFFFFFF. Change the address specified by the start option.

### **L3101 (F) Section “section 1” overlaps section “section 2”**

The addresses of **section 1** and **section 2** overlap. Change the address specified by the start option.

### **L3102 (F) Section contents overlap in absolute section “section”**

Data addresses overlap within an absolute address section. Modify the source program.

### **L3110 (F) Illegal cpu type “cpu type” in “file”**

A file with a different cpu type was input.

### **L3111 (F) Illegal encode type “endian type” in “file”**

A file with a different endian type was input.

### **L3112 (F) Invalid relocation type in “file”**

There is an unsupported relocation type in **file**. Ensure the compiler and assembler versions are correct.

### **L3200 (F) Too many sections**

The number of sections exceeded the limit. It may be possible to eliminate this problem by specifying multiple file output.

### **L3201 (F) Too many symbols**

The number of symbols exceeded the limit. It may be possible to eliminate this problem by specifying multiple file output.

### **L3202 (F) Too many modules**

The number of modules exceeded the limit. Divide the library.

**L3300 (F) Cannot open file : “file”****P3300**

**file** cannot be opened. Check whether the file name and access rights are correct.

**L3301 (F) Cannot close file : “file”**

**file** cannot be closed. There may be insufficient disk space.

**L3302 (F) Cannot write file : “file”**

Writing to **file** is not possible. There may be insufficient disk space.

**L3303 (F) Cannot read file : “file”****P3303**

**file** cannot be read. An empty file may have been input, or there may be insufficient disk space.

**L3310 (F) Cannot open temporary file****P3310**

A temporary file cannot be opened. Check to ensure the HLNK\_TMP specification is correct, or there may be insufficient disk space.

**L3311 (F) Cannot close temporary file**

A temporary file cannot be closed. There may be insufficient disk space.

**L3312 (F) Cannot write temporary file**

Writing to a temporary file is not possible. There may be insufficient disk space.

**L3313 (F) Cannot read temporary file**

A temporary file cannot be read. There may be insufficient disk space.

**L3314 (F) Cannot delete temporary file**

A temporary file cannot be deleted. There may be insufficient disk space.

**L3320 (F) Memory overflow****P3320**

There is no more space in the usable memory within the linkage editor. Increase the amount of memory available.

**L3400 (F) Cannot execute “load module”**

**load module** cannot be executed. Check whether the path for **load module** is set correctly.

**L3410 (F) Interrupt by user**

An interrupt generated by (cntrl) + C keys from a standard input terminal was detected.

**L3420 (F) Error occurred in “load module”.**

An error occurred while executing the **load module**.

**P3500 (F) Bad instantiation request file -- instantiation assigned to more than one file**

There was an error in the instantiation request file. Recompile the linked files.

**P3501 (F) Instantiation loop**

There is a loop in the instantiation processing. An input file name may coincide with an instantiation request file in another file. Change the file names so that when the extension is removed they do not coincide.

**P3502 (F) Cannot create instantiation request file “file”**

The instantiation request file cannot be created. Check whether access rights for the object creation directory are correct.

**P3503 (F) Cannot change to directory “directory”**

The current directory cannot be changed to **directory**. Check to ensure that **directory** exists.

**P3504 (F) File “file” is read-only**

**file** is a read-only file. Change the access rights.

**L4000 (–) Internal error : (“internal error code”) “file line number” / “comment”**

**P4000**

An internal error occurred during processing by the optimizing linkage editor. Make a note of the internal error number, file name, line number, and comment in the message, and contact the support department of the vendor.



# Section 15 Error Messages for the Standard Library Generator and Format Converter

## 15.1 Error Format and Error Levels

In this section, error messages output in the following format and the details of errors are explained.

Error number            (Error level) Error message

Error details

There are three different error levels, corresponding to different degrees of seriousness.

Error Number	Error Level	Error Type	Description
G1000–G1999	(W)	Warning	Processing is continued.
G2000–G2999	(E)	Error	Option analysis processing is continued; processing is interrupted.
G3000–G3999	(F)	Fatal	Processing is interrupted.

## 15.2 List of Messages

### G1001 (W) Debug information ignored

Functions both with and without optimization as specified by **#pragma option** exist in the file to be converted. Conversion will take place without including the debugging information.

### G1002 (W) Command parameter specified twice

An option has been specified more than once. Only the last of the specifications is effective. Check the specifications of the options.

### G2001 (W) Cannot open file “file”

Cannot open file. Check the file name and access rights.

### G2002 (E) Illegal file type “file”

A file other than an object file or a library file has been specified for conversion from SYSROF to ELF. A file other than a load module file has been specified for conversion from ELF to SYSROF. Check the file type and re-execute.

### G2003 (E) Illegal file format “file”

The file format is invalid. Check the file’s contents and re-execute.

**G3001 (F) Invalid command parameter “parameter”**

An invalid command parameter has been specified. Check the command parameter, and re-execute.

**G3002 (F) No input file**

No input file was found.

**G3003 (F) Command parameter buffer overflow**

The command line exceeds 32767 characters.

**G3101 (F) Cannot open file “file”**

Cannot open file. Check the file name and access rights.

**G3102 (F) Cannot input file “file”**

Cannot input from the specified file. Check whether a file to be converted is accessed or not.

**G3103 (F) Cannot create file “file”**

Cannot create a file. Check the available disk space.

**G3104 (F) Cannot output file “file”**

Cannot write to a file. Clear the write prohibition.

**G3105 (F) Cannot open internal file**

Cannot open the temporary file which has been generated internally. Check that the temporary file is not being accessed.

**G3106 (F) Cannot output internal file**

Cannot output to the temporary file which has been generated internally. Check the disk space, or check the disk for a physical error.

**G3107 (F) Memory overflow**

The required memory area for internal use cannot be allocated. Reserve the necessary amount of memory and re-execute.

**G3108 (F) Illegal format in archive “file”**

The specified file is not in an archive format.

**G3109 (F) Cannot find “file name”**

Cannot find the file. Check the settings of the environment variable PATH.

**G3201 (F) Cannot execute compiler**

Cannot initiate the compiler. Check the path name and the environment variables of the compiler.

**G3202 (F) Cannot execute optlinker**

Cannot initiate the optimizing linkage editor. Check the path name of the optimizing linkage editor.

**G3203 (F) Interrupt by user**

An interrupt has been detected during execution.

**G3204 (F) Cannot execute assembler**

Cannot initiate the assembler. Check the path name of the assembler.

**G3300 (F) Already existent file “file”**

The file already exists.



# Section 16 Limitations

## 16.1 Limitations of the Compiler

Table 16.1 shows the limits of the compiler.

Source programs must fall within these limits.

**Table 16.1 Limitations of the Compiler**

Classification	Item	Limit
Invoking the compiler	Total number of macro names that can be specified using the <b>define</b> option	None
	Length of file name (characters)	None (depends on the OS)
Source programs	Length of one line (characters)	32,768 (H8SX/H8S) or 16,384 (300H/300)
	Number of source program lines in one file	None
	Number of source program lines that can be compiled	None
Preprocessing	Nesting levels of files in an <b>#include</b> directive	None
	Total number of macro names in a <b>#define</b> directive	None
	Number of parameters that can be specified using a macro definition or a macro call operation	None
	Number of macros that can be replaced	None
	Nesting levels of an <b>#if</b> , <b>#ifdef</b> , <b>#ifndef</b> , <b>#else</b> , or <b>#elif</b> directive	None
	Total number of operators and operands that can be specified in an <b>#if</b> or <b>#elif</b> directive	None
Declarations	Number of function definitions	None
	Number of external identifiers used for external linkage	None
	Number of valid internal identifiers used in one function	None
	Total number of pointers, arrays, and functions that qualify the basic type	16
	Array dimensions	6

**Table 16.1 Limitations of the Compiler (cont)**

Classification	Item	Limit
Declarations	Size of arrays and structures* <sup>1</sup>	
	<ul style="list-style-type: none"> <li>H8SX: Normal mode</li> <li>H8S/2600: Normal mode</li> <li>H8S/2000: Normal mode</li> <li>H8/300H: Normal mode</li> <li>H8/300</li> </ul>	65,535 bytes
	<ul style="list-style-type: none"> <li>H8SX: Middle mode</li> <li>H8SX: Advanced mode (with ptr16 option)</li> <li>H8SX: Maximum mode (with ptr16 option)</li> </ul>	32,767 bytes
	<ul style="list-style-type: none"> <li>H8/300H: Advanced mode</li> </ul>	16,777,215 bytes
	<ul style="list-style-type: none"> <li>H8SX: Advanced mode (without ptr16 option)</li> <li>H8SX: Maximum mode (without ptr16 option)</li> <li>H8S/2600: Advanced mode</li> <li>H8S/2000: Advanced mode</li> </ul>	2,147,483,647 or 4,294,967,295 (if <b>legacy=v4</b> is specified) bytes
Statements	Nesting levels of compound statements	None
	Nesting levels of combinations of iterative statement ( <b>while</b> , <b>do</b> , or <b>for</b> statement) and selective statement ( <b>if</b> or <b>switch</b> statement)	4,096 (H8SX/H8S) or 256 (300H/300)
	Number of <b>goto</b> labels that can be specified in one function	2,147,483,646 (H8SX/H8S) or 511 (300H/300)
	Number of <b>switch</b> statements	2,048* <sup>4</sup>
	Nesting levels of <b>switch</b> statements	2,048 (H8SX/H8S) or 128 (300H/300)
	Number of <b>case</b> labels in a single <b>switch</b> statement	2,147,483,646 (H8SX/H8S) or 511 (300H/300)
	Nesting levels of <b>for</b> statements	2,048 (H8SX/H8S) or 128 (300H/300)
Expressions	Length of string literal	32,766
	Number of parameters that can be specified using a function definition or a function call operation	2,147,483,646 (H8SX/H8S) or 63 (300H/300)* <sup>2</sup>
	Total number of operators and operands that can be specified in one expression	About 500
Standard library	Number of files that can be opened simultaneously in an <b>open</b> function	Variable* <sup>3</sup>

Notes: If the **legacy=v4** option is specified, the limits will be the same as those of 300H/300.

1. When the bit width of the address space is specified in advanced, middle or maximum mode, the address space size corresponding to the specified bit width is given priority. The ptr16 option changes the limit of H8SX advanced or maximum mode.
2. For nonstatic function members, 62.
3. For details, refer to section 9.2.2 (5), C/C++ library function initial settings (\_INITLIB).
4. The items in which the limitation is changed by this version.

## 16.2 Limitations of the Assembler

Table 16.2 shows the limits of the assembler.

**Table 16.2 Limitations of the Assembler**

Item		Limit
Length of one line (characters)		8192
Character constants		Up to 4
Length of symbol		None <sup>*1</sup>
Number of symbols		None
Number of externally referenced symbols		None
Number of externally defined symbols		None
Maximum size for a section <sup>*2</sup>	H8SX in maximum mode	Up to H'FFFFFFFF bytes
	H8SX in advanced mode	Up to H'FFFFFFFF bytes
	H8SX in middle mode	Up to H'00FFFFFF bytes
	H8SX in normal mode	Up to H'0000FFFF bytes
	H8S/2600 in advanced mode	Up to H'FFFFFFFF bytes
	H8S/2600 in normal mode	Up to H'0000FFFF bytes
	H8S/2000 in advanced mode	Up to H'FFFFFFFF bytes
	H8S/2000 in normal mode	Up to H'0000FFFF bytes
	H8/300H in advanced mode	Up to H'00FFFFFF bytes
	H8/300H in normal mode	Up to H'0000FFFF bytes
	H8/300	Up to H'0000FFFF bytes
	H8/300L	Up to H'0000FFFF bytes
Number of sections	When gooptimize is specified:	With debug: H'FEF1
		Without debug: H'FEFA
	When gooptimize is not specified:	With debug: H'FEF2
		Without debug: H'FEFB
File include		Up to 30 levels of nesting
Length of string literal (characters)		Up to 255
Length of file name (characters)		None (depends on the OS)

- Notes: 1. For a preprocessor variable name, macro name, and macro parameter name, it is limited to 32 characters.  
There is no limitation on the number of characters in a replacement symbol specified in `-DEFINE` or `.DEFINE`. However, the replacement string literal is limited to 255 characters, and up to 8192 characters can be specified in one line.
2. The maximum size of a section differs according to the specified address space.



## 17.1 Compiler Functions

### 17.1.1 Overview

This section shows the usage of the functions supporting AE5.

In order to use the intrinsic functions supported by the compiler to access the EEPROM with the EEPMOV.B or EEPMOV/P.W instruction, include the header file <machine.h> or <eeprom.h> and specify the EEPROM option. Also, specify the EEPROM option if the EEPMOV/P.W instruction is written in the `__asm{ }` block (except for specification of **-cpu=ae5**). To specify the EEPROM option in HEW, write “-EEPROM” or “-eeprom” in the edit box of [User specified options :] in the <Other> category of the option window’s C/C++ tab.

### 17.1.2 Compiler Options

The following describes options added for V6.01. In addition, the compiler can use other options and expanded functions that are available when the CPU type is H8SXA (except for the intrinsic function `set_vbr`).

**Table 17.1 Special Options**

Item	Command Line Format	Dialog Menu	Specification
CPU type	cpu=ae5	C/C++ <CPU>	Specifies the CPU.
Use of EEPMOV/P.W	eeprom	C/C++ <Other> [User defined options :]	Allows use of EEPMOV/P.W.

C/C++ &lt;CPU&gt;

Description Format: CPu = AE5

Description: Specifies the CPU type of the object program to be generated.  
 When AE5 is specified for the suboption, it is impossible to specify a multiplier and/or a divider.  
 The address space size is fixed to 24 bits.  
 If this option is specified, the **eeeprom** option is always valid.

**EEPROM:****Access to EEPROM**

C/C++ &lt;Other&gt;[User defined options :]

Description Format: EEPROM

Description: Allows the use of the intrinsic functions using the EEPMOV.B or EEPMOV/P.W instruction to access the EEPROM. If this option is specified and if the header file <eeprom.h> is included, the following intrinsic functions are expanded to the EEPMOV.B or EEPMOV/P.W instruction with update of ECR or EPR.

- eeepromb
- eeepromw
- eeepromb\_epr
- eeepromw\_epr

Also, this option allows use of the EEPMOV/P.W instruction in the `__asm{ }` block.

Remarks:

1. For the details of the intrinsic functions, refer to the description of each function.
2. Refer to the hardware manual for the details of ECR and EPR.

### 17.1.3 Intrinsic Functions

**Table 17.2 Intrinsic Functions**

Item	Specification	Function
Special instructions	unsigned char eepromb( void *dst, const void *src, unsigned char size, volatile unsigned char *ecr, unsigned char ecrval)	EEPMOV.B transfers a memory block, whose byte amount is specified by size, from the address specified by src to the address specified by dst after updating ECR.
	unsigned int eepromw( void *dst, const void *src, unsigned int size, volatile unsigned char *ecr, unsigned char ecrval)	EEPMOV.P.W transfers a memory block, whose byte amount is specified by size, from the address specified by src to the address specified by dst after updating ECR.
	unsigned char eepromb_epr( void *dst, const void *src, unsigned char size, volatile unsigned char *ecr, unsigned char ecrval, volatile unsigned char *epr, unsigned char eprval)	EEPMOV.B transfers a memory block, whose byte amount is specified by size, from the address specified by src to the address specified by dst after updating EPR and ECR.
	unsigned int eepromw_epr( void *dst, const void *src, unsigned int size, volatile unsigned char *ecr, unsigned char ecrval, volatile unsigned char *epr, unsigned char eprval)	EEPMOV.P.W transfers a memory block, whose byte amount is specified by size, from the address specified by src to the address specified by dst after updating EPR and ECR.

**unsigned char eepromb (void \*dst, const void \*src, unsigned char size,  
volatile unsigned char \*ecr, unsigned char ecrval)**  
**unsigned int eepromw (void \*dst, const void \*src, unsigned int size,  
volatile unsigned char \*ecr, unsigned int ecrval)**  
**: Block Transfer Instructions (with ECR Setting)**

**Description:** A memory block, whose size is shown by **size**, is transferred from the address specified by **src** to the address specified by **dst**. The **eepromb** intrinsic function transfers a memory block with the EEPMOV.B instruction, and **eepromw** with the EEPMOV.P.W instruction. These intrinsic functions sets **dst**, **src** and **size** to the registers, sets **ecrval** to the address pointed by **ecr**, and then transfers the memory block. If transfer completes successfully, 0 is returned. If transfer fails, the remaining size of the memory block left is returned. **size** of **eepromb** can take 0 to 255, and **size** of **eepromw** can take 0 to 65535. However, if size is 0, no transfer occurs.

**Header:** <machine.h>/<eeprom.h>

**Return Values:** Size of data that was left without transfer (0 to size)

**Parameters:**

dst	Pointer to the destination
src	Pointer to the source
size	Transfer size
ecr	Address of hardware register ECR
ecrval	Value to be set to hardware register ECR

**Example:**

```
#include <eeprom.h>
#define ecr_ptr ((volatile unsigned char *) (0xZZZZZZ))
char a[10], b[10];
unsigned char x;
void f(void)
{
    x = eepromb(b, a, 10, ecr_ptr, 1);
}
```

**Remarks:**

1. This intrinsic function is valid only when the CPU type is AE5 or H8SX and the **-eeprom** option is specified.
2. To use these intrinsic functions by specifying the CPU type other than AE5, the **eeprom** option must be specified at compilation.
3. Refer to the hardware manual for the details of ECR, EPR, and other related issues.

**unsigned char eepromb\_epr** (void \*dst, const void \*src, unsigned char size,  
volatile unsigned char \*ecr, unsigned char ecrval,  
volatile unsigned char \*epr, unsigned char eprval)  
**unsigned int eepromw\_epr** (void \*dst, const void \*src, unsigned int size,  
volatile unsigned char \*ecr, unsigned char ecrval,  
volatile unsigned char \*epr, unsigned char eprval)

**: Block Transfer Instructions (with EPR and ECR Setting)**

**Description:** A memory block, whose size is shown by **size**, is transferred from the address specified by **src** to the address specified by **dst**. The **eepromb\_epr** intrinsic function transfers a memory block with the EEPMOV.B instruction, and **eepromw\_epr** with the EEPMOV.P.W instruction. These intrinsic functions set **dst**, **src** and **size** to the registers, set **eprval** to the address pointed by **epr**, set **ecrval** to the address pointed by **ecr**, and then transfer the memory block. If transfer completes successfully, 0 is returned. If transfer fails, the remaining size of the memory block left is returned. **size** of **eepromb\_epr** can take 0 to 255, and **size** of **eepromw\_epr** can take 0 to 65535. However, if size is 0, no transfer occurs.

**Header:** <machine.h>/<eeprom.h>

**Return Values:** Size of data that was left without transfer (0 to size)

**Parameters:**

dst	Pointer to the destination
src	Pointer to the source
size	Transfer size
ecr	Address of hardware register ECR
ecrval	Value to be set to hardware register ECR
epr	Address of hardware register EPR
eprval	Value to be set to hardware register EPR

**Example:**

```
#include <eeprom.h>
#define ecr_ptr ((volatile unsigned char *) (0xZZZZZZ))
#define epr_ptr ((volatile unsigned char *) (0xWWWWWW))
char a[10], b[10];
unsigned char x;
void f(void)
{
    x = eepromb_epr(b, a, 10, ecr_ptr, 1, epr_ptr, 1);
}
```

**Remarks:**

1. This intrinsic function is valid only when the CPU type is AE5 or H8SX and the **-eeprom** option is specified.
2. To use these intrinsic functions by specifying the CPU type other than

- AE5, the **eeeprom** option must be specified at compilation.
3. Refer to the hardware manual for the details of ECR, EPR, and other related issues.

## 17.2 Assembler Functions

The following describes options added for V6.01. In addition, the compiler can use other options and expanded functions that are available when the CPU type is H8SXA. However, instructions with SBR and VBR cannot be used.

**Table 17.3 Special Options**

Item	Command Line Format	Dialog Menu	Specification
CPU type	-CPu = AE5	Assembly <CPU>	Specifies the CPU.
Use of EEPMOV/P.W	-EEPROM	Assembly <Other> [User defined options :]	Allows use of EEPMOV/P.W.

**CPu:** **CPU Type**

Assembly <CPU>

Description Format: CPu = AE5

Description: Specifies the CPU type of the object program to be generated.  
 When AE5 is specified for the suboption, it is impossible to specify a multiplier and/or a divider.  
 The address space size is fixed to 24 bits.  
 If this option is specified, the **eeeprom** option is always valid.

## EEPROM:

## Access to EEPROM

Assembly <Other>[User defined options :]

Description Format: EEPROM

Description: Allows the use of the EEPMOV/P.W instruction.

Remark: This option is valid only when the CPU type is H8SX.





# Section 18 Notes on Version Upgrade

## 18.1 Notes on Version Upgrade

This section contains notes that apply when the version is upgraded from an earlier version (H8S, H8/300 Series C/C++ Compiler Package: Ver. 3.x or earlier).

### 18.1.1 Guaranteed Program Operation

When a program is developed with an upgraded compiler version, operation of the program may change. When creating the program, note the following and sufficiently test your program.

#### 1. Programs Depending on Execution Time and Timing

C/C++ language specifications do not specify the program execution time. Therefore, a version difference in the compiler may cause operation changes due to timing lag of the program execution time with peripherals such as the I/O, or may cause processing time differences in asynchronous processing such as in interrupts.

#### 2. Programs Including an Expression with Two or More Side Effects

Operations may change depending on the compiler version when two or more side effects are included in one expression.

Example

```
a[i++] = b[i++];    /* Increment order of i is undefined. */  
f(i++, i++);        /* Parameter value changes according to increment order. */  
/* This results in f(3, 4) or f(4, 3) when the value of i is 3. */
```

#### 3. Programs with Overflow Results or an Illegal Operation

The value of the result is not guaranteed when an overflow occurs or an illegal operation is performed. Operation of the program may change depending on the compiler version.

Example

```
int a, b;  
x = (a * b) / 10;    /* This may cause an overflow depending on the value range of  
a and b. */
```

#### 4. No Initialization of Variables and Type Inequality

When a variable is not initialized or the parameter and return value types do not match between the calling and called functions, an illegal value is accessed. Operation of the program may change depending on the compiler version.

Example

file 1:

```
int f(double d)
{
    :
}
```

file 2:

```
int g(void)
{
    f(1);
}
```

The parameter of the caller file is the int type, but the parameter of the function-defining file is the double type. Therefore, a value cannot be correctly referenced.

The information provided here does not include all cases that may occur. Please use this compiler prudently, and sufficiently test your programs keeping the differences between the compiler versions in mind.

#### 18.1.2 Compatibility with the Earlier Version

The following notes cover situations in which the compiler is used to generate a file that is to be linked with object or library files generated by the Ver. 4.0 or earlier compiler and the accompanying assembler or linkage editor, or in case the debugger created for the version 3.x or earlier is used as it is.

##### 1. **strict\_ansi** (from Ver. 6.01)

When **strict\_ansi** is specified, the results of floating-point operations may differ from those produced by earlier versions of the compiler (Ver. 4.x or earlier). To obtain uniform results, omit the **strict\_ansi** option from the compilation command, or recompile all files with the **strict\_ansi** option.

## 2. **cpuexpand (from Ver. 6.01)**

If **cpuexpand** is specified, the result of some operations may differ from those produced by earlier versions of the compiler (Ver.4.x or earlier). To obtain uniform results, specify **legacy=v4** for compilation when the CPU type setting is 2600A, 2600N, 2000A, or 2000N, or specify **cpuexpand** for all files and recompile them when the CPU type setting is any of the H8SX variants. For the expressions that produce variable results, refer to the description of **cpuexpand=v6** in section 2.2.2, Object Options.

## 3. **code=asmcode (from Ver. 6.01)**

From this version (Ver.6.01), the compiler outputs a **.STACK** directive within the assembly-source program if **code=asmcode** is specified. Thus the assembler for use must be Ver.6.01, which supports **.STACK**.

## 4. **Changed Section for Explicitly Initialized Variables (from Ver. 6.01)**

In Ver.6.00.00, explicitly initialized variables for which H8/300, H8/300H, H8S/2000, or H8S/2600 had been specified as the CPU type were output to section D.

In Ver.6.00.01, if the CPU type is H8S/2000 or H8S/2600, explicitly initialized variables are output to section C. If the CPU type is H8/300 or H8/300H, explicitly initialized variables are still output to section D. With the H8SX setting, however, explicitly initialized variables are always output to section C, regardless of the compiler version.

## 5. **Object Format (from Ver. 4.0)**

The object file format has been changed from SYSROF to the standard format ELF. The debugging information format has also been changed to the standard format DWARF2.

Before an object file (SYSROF) output by Ver. 3.x or earlier of the compiler or assembler is to input to the latest optimizing linkage editor, use a file converter to convert it to the ELF format. However, relocatable files output by the linkage editor (extension: rel) and library files that include one or more relocatable files cannot be converted.

When a debugger which supports the SYSROF or ELF/DWARF1 format load modules is used, use the file converter to convert the load module from the ELF/DWARF2 format to the SYSROF or ELF/DWARF1 format. However, the debugging information will not be converted and only the object part will be valid if **#pragma option** (new feature of Ver. 4.0 compiler) has created a file in which a function with optimization and that without optimization coexist.

## 6. **Added an Option to Modify the Function Interface (from Ver. 4.0)**

Options **structreg** and **longreg** have been added to modify the function interface rules. Recompile all files after you have specified either option. Modify the interfaces of assembly routines, too.

## 7. Stack Area (from Ver. 4.0)

Option **stack** can be used to specify the size used in calculation of the stack area size.

When this option is omitted, **stack=medium** (stack calculation will be performed only in least significant 2 bytes) will be assumed. To change this, specify another size by using option **stack**.

## 8. const Data Output Section (from Ver. 4.0)

In Ver. 3.x, variables in a **const** declaration were output to section D. In Ver. 4.0 and later those variables are output to section C.

## 9. Data Allocation (from Ver. 4.0)

The options **align/noalign** can be used to rearrange data according to boundary alignment.

When this option is omitted, **align** is assumed, and data is grouped by boundary alignment. To inhibit rearrangement, specify **noalign**.

## 10. Boundary Alignment of Sections \$ABS8C, \$ABS8D, and \$ABS8B (from Ver. 4.0)

The boundary alignment value for sections \$ABS8C, \$ABS8D, and \$ABS8B that are output when **#pragma abs8**, **\_\_abs8**, or option **abs8** are specified has been changed from 2 to 1.

Accordingly, variables that are affected by **#pragma abs8**, **\_\_abs8** or option **abs8** have been changed from:

- Variables or arrays that have char or unsigned char type or
- Structures or classes that have char or unsigned char type variables or arrays as members to:
- Variables, arrays, structures, or classes whose the boundary alignment value is 1.

## 11. Point of Origin for Locating Include Files (from Ver. 4.0)

In the new version, option **chgincpath** has been abolished. When an include file that has been specified with a relative path is searched for, the search starts from the directory that contains the source file.

## 12. C++ Program (from Ver. 4.0)

Since the encoding rule and execution method were changed, C++ object files created by the earlier version of the compiler cannot be linked. Be sure to recompile such files.

The names of the library functions for initial/post processing of the global class object, which are used to set the execution environment, have also been changed. Refer to section 9.2.2, Execution Environment Settings, and modify the name.

## 13. Abolition of Common Section (Assembly Program, from Ver. 4.0)

With the change of the object format, support for of common section has been abolished.

## 14. Specification of Entry via .END Directive (Assembly Program, from Ver. 4.0)

Only an externally defined symbol can be specified as entry to the **.END** directive.

## 15. Inter-module Optimization (from Ver. 4.0)

Object files output by the old version of the compiler or the assembler are not targeted for inter-module optimization. Be sure to recompile and reassemble such files so that they are targeted for inter-module optimization as required.

## 16. Objects Supported by the Optimizing Linkage Editor

The optimizing linkage editor supports different compiler or assembler depending on the version. The following shows the version of the supported tool. Linkage processing for the object file that is not described is not guaranteed.

- Optimizing linkage editor Ver. 7: Ver. 4.0 or lower of the compiler, Ver. 4.0 or lower of the assembler
- Optimizing linkage editor Ver. 8: Ver. 6.00 or lower of the compiler, Ver. 6.00 or lower of the assembler
- Optimizing linkage editor Ver. 9.00: Ver. 6.01 or lower of the compiler, Ver. 6.01 or lower of the assembler

## 17. Option Consistency

The following compiler options should be the same among the earlier versions and Ver. 6.01.  
cpu, regparam, pack, structreg/nostructreg, longreg/nolongreg, stack, double=float, rtti, exception

However, to link object files generated by earlier versions than Ver. 6.01, the following options newly added to Ver. 6.01 should not be used.

structreg, longreg, stack=small/medium, double=float, rtti, exception

Also, to link object files generated by earlier versions than Ver. 4.0, the following options newly added to Ver. 4.0 should not be used.

structreg, longreg, stack=small/medium, double=float, rtti, exception

Also, to link object files generated by earlier versions than Ver. 3.0, the following options newly added to Ver. 3.0 should not be used.

regparam=3, pack=1

## 18.1.3 Command-line Interface

### 1. How to Specify Assembler (Ver. 4.0) and Optimizing Linkage Editor (Ver. 7.0) Command Lines

Spaces must be inserted between file names and options.

There are no limitations on the order in which options and their associated file names are specified.

## 2. Optimizing Linkage Editor Option (from Ver. 7.0)

Support for the interactive specification of options has been abolished.

The inter-module optimizing tool (optlnk38), linkage editor (lnk), librarian (lbr), and object converter (cnvs) of earlier versions have been integrated into optimizing linkage editor (optlnk). Accordingly, specifications have changed significantly. Tables 18.1 and 18.2 list the changed commands.

**Table 18.1 Changed Linkage Commands**

No.	Command Name	Ver. 6.0	Ver. 7.0	Note
1	start	start = section (address) Abbreviation: st	start = section/address Abbreviation: star	—
2	rom	rom = (rom section, ram section)	rom = rom section/ ram section	—
3	define	define = external name (defined value)	define = external name = defined value	—
4	rename	rename = ed = before change (after change), er = before change (after change), un = before change (after change) Abbreviation: re	rename = (before change = after change), (before change = after change), — Abbreviation: ren	The conception of unit has been abolished due to the change in the object format.
5	delete	delete = ed = unit.symbol un = unit	delete = (symbol) —	The conception of unit has been abolished due to the change in the object format.
6	print/noprint	print noprint	list —	File name can be omitted.
7	mlist	mlist	list	—
8	information	information	message	—
9	directory	directory	HLNK_DIR (environment variable)	—
10	form	Abbreviation: f	Abbreviation: fo	—
11	output/nooutput	Abbreviation: o; nooutput can be specified.	Abbreviation: ou; nooutput cannot be specified.	Only output can be specified.

**Table 18.1 Changed Linkage Commands (cont)**

No.	Command Name	Ver. 6.0	Ver. 7.0	Note
12	cpu	Abbreviation: c	Abbreviation: cp	Ranges range can be directly.
13	elf/sysrof/sysroflplus	elf/sysrof/sysroflplus	Abolished	Always ELF
14	exclude/noexclude	exclude/noexclude	Abolished	Always exclude
15	align_section	align_section	Abolished	Always valid*
16	check_section	check_section	Abolished	Always valid*
17	cpucheck	cpucheck	Abolished	Always valid*
18	udf/noudf	udf/noudf	Abolished	Always output*
19	udfcheck	udfcheck	Abolished	Always valid*
20	echo/noecho	echo/noecho	Abolished	Always restricted
21	exchange	exchange	Abolished	The conception of unit has been abolished due to the change in the object format.
22	autopage	autopage	Abolished	No target cpu
23	abort	abort	Abolished	Interactive mode has been abolished.
24	list	list	Abolished	Different from the list option of V7.
25	library/nolibrary	nolibrary can be specified.	nolibrary cannot be specified.	Only library can be specified.
26	exit	Cannot be omitted.	Can be omitted.	—
27	debug/nodebug	At default: nodebug	At default: Depends on the debugging information in the input file	—

Note: Can be invalidated by the **change\_message** option.

**Table 18.2 Changed Librarian Commands**

No.	Command Name	Ver. 2.0	Ver. 7.0	Note
1	add	add	input	—
2	directory	directory	HLNK_DIR (environment variable)	—
3	slist	slist	list show	—
4	list	list (s)	list show	—
5	delete	Abbreviation: d	Abbreviation: del	—
6	create	create (s   u)	library form = library (s   u)	—
7	output	output (s   u)	output form = library (s   u)	—
		Abbreviation: o	Abbreviation: ou	
8	replace	Abbreviation: r	Abbreviation: rep	—
9	abort	abort	Abolished	Interactive mode has been abolished.
10	exit	Cannot be omitted.	Can be omitted.	—

### 18.1.4 Provided Contents

In the H8S, H8/300 Series C/C++ Compiler Package, the following files have been changed from Ver. 4.0 package.

#### 1. CPU Information Analyzer

In the new version, an address range can be specified directly with the `cpu` option of `optlink`.

An old-version `cpu` information file can also be used in the new version. To modify or create CPU information, specify the address range directly with the **cpu** option.

#### 2. Standard Library File

In order to choose a function interface and optimizing options, a standard library generator is provided instead of the conventional standard library file.

#### 3. Header File

`defbool.h` has been abolished because the `bool` type has been supported in the new version.



## 18.1.5 List File Specification

### 1. Compile Listing (from Ver. 4.0)

The layout of the columns is changed to look better. Also, the number of columns of a tab can be selected.

### 2. Optimizing Linkage Editor (from Ver. 7.0)

The formats of the conventional linkage map and library list have been renewed.

## 18.2 Additions and Improvements

### 18.2.1 Common Additions and Improvements

#### 1. Added and Improved Features in Compiler Ver. 4.0, Assembler Ver. 4.0 and Optimizing Linkage Editor Ver. 7.0

##### a. Loosening Limits on Values

Limitations on source programs and command lines have been greatly loosened:

- Length of file name: 251 bytes -> unlimited
- Length of symbol: 251 bytes -> unlimited
- Number of symbols: 65,535 -> unlimited
- Number of source program lines: C/C++: 32,767, ASM: 65,535 -> unlimited
- Length of C program line: 8,192 characters -> 16,384 characters
- Length of C program string literal: 512 characters -> 16,384 characters
- Length of subcommand file lines: ASM: 300 bytes, optlnk: 512 bytes -> unlimited
- Number of parameters of the optimizing linkage editor **rom** option: 64 -> unlimited

##### b. Hyphens for Directory and File Names

A hyphen (-) can be specified for directory or file names.

##### c. Specification of Copyright Display

Specifying the **logo/nologo** option can specify whether or not the copyright banner is displayed.

##### d. Prefix to Error Messages

To support the error-help function in the HEW, a prefix has been added to error messages for the compiler and optimizing linkage editor.

## 18.2.2 Added and Improved Compiler Features

### 1. Added and Improved Features in Ver. 4.0

#### a. Use of Keyword

Attributes can be specified in declarations or definitions of functions or variables by using keywords (`_interrupt`, `_indirect`, `_entry`, `_abs8`, `_abs16`, `_regsave`, `_noregsave`, `_inline`, or `_global_register`).

#### b. Creation of Vector Table

The vector table of functions can be created automatically when `vect` is specified by `#pragma interrupt`, `#pragma indirect`, `#pragma entry`, `_interrupt`, `_indirect`, or `_entry`.

#### c. Support of `_evenaccess`

Memory access of a variable at even-numbered byte boundary is guaranteed with `_evenaccess` specified.

#### d. Expanded Register Parameter Specification

`_regparam2` and `_regparam3` can be used to specify the number of register parameters in a function.

#### e. Specifying Options in Function Units

Options can be specified on function by function basis by using `#pragma option`.

#### f. Confining address calculation rang of aggregates.

Optimizes address calculation code of arrays or structures by using `_near8` or `_near16`. However, the pointer size is not changed.

#### g. Confining address calculation rang of the stack

Optimizes stack address calculation code of stack areas by using `stack`.

#### h. Added Intrinsic Functions

The following intrinsic functions were added.

— Unsigned overflow operations

#### i. Supporting `double=float`

In the new version, `double=float` can be specified so that data declared as double-precision type and floating-point constants are both regarded as float type.

#### j. Strengthening `noregsave` Feature

When a function declared with `#pragma noregsave` or `_noregsave` is called, the contents of the registers are guaranteed by the calling function.

**k. Specifying Multiple Sets of Include Directory by Using Environment Variables**

Multiple include directories can be specified by using the include directory environment variable (CH38).

**l. Allocating Structure Parameter or Return Value to Register**

Option **structreg** is used to allocate a small-size structure parameter or return value to a register.

**m. Allocating 4-Byte Parameter or Return Value to Register (cpu=300)**

Option **longreg** is used to allocate a 4-byte parameter or return value to a pair of registers.

**n. Conditions for Moving a Non-volatile Variable Outside a Loop**

A non-volatile external variable in an iteration condition inhibits external variable optimization from moving the non-volatile external variable out of the loop even though there are no side effects (function calls or assignment expressions) in an iteration condition.

**o. Support of speed=loop=1|2**

Option **speed=loop=1|2** controls optimization of loop expansion.

**p. Modifying Data Allocation by Boundary Alignment**

Data can be reallocated for each boundary alignment so that gaps that are generated by the boundary alignment are minimized.

**q. Added Implicit Declarations**

`__HITACHI__` and `__HITACHI_VERSION__`, are implicitly declared by `#define`.

**r. static Label Name**

The specification of label names as references to static labels, which has file scope, `#pragma asm` and `#pragma endasm`, and `#pragma inline_asm` has been changed to `__$ (name)`.

However, in a linkage list, the name is displayed as `_ (name)`.

**s. Extension and Change of Language Specifications**

— Inhibits errors in initializing unions.

Example:

```
union{
    char c[4];
}uu={ { 'a', 'b', 'c' } };
```

— enum can be used as bit fields.

Example:

```
struct{
    enum E1{a,b,c}m1:2;
    enum E1      m2:2;
};
```

— Inhibits the output of an error message when a comma “,” is written after the last enumeration member.

Example:

```
enum E1{a,b,c,}m1;
```

— A union can be declared with an initial value in a single declaration.

Example:

```
union U{
    int a,b;
}u1;
void test(){
    union U u2 = u1;
```

— Loosened the level of checking for errors in casting of symbol address expressions at C compilation.

Example:

```
int x;
short addr1=(short)&x;
```

— Loosened the restrictions on the order of writing declaration of functions and variables, and #pragma declarations in C programs.

Example:

```
void f(void);
#pragma interrupt f
void f(void){}    //#pragma declaration following function declaration is valid.
                  //(In Ver. 3.0, an error would have occurred.)
```

— Modifies the restrictions on the order of writing declarations of functions and variables, and `#pragma` declarations in C++ programs.

Example:

```
void f(void){}
#pragma interrupt f //pragma declaration following function definition is ineffective.
void f(void);      //An error will occur when a #pragma declaration modifying the
                  //following function declaration follows a function definition.
```

— Supports exception and template according to the C++ language specifications.

## 2. Added and Improved Features in Upgrade from Ver. 4.0 to Ver. 6.0

(Note: Ver. 5.0 does not exist and is a missing number.)

### a. Support for New CPU

Creation of an object file with a CPU type of H8SX is supported.

### b. Support for 2-byte Pointer (only in H8SX)

The `__ptr16` keyword or option `ptr16` can be used to specify use of a 2-byte pointer. They are valid in H8SX advanced mode or H8SX maximum mode.

### c. Specifying Bit Field Order

`#pragma bit_order` or the `bit_order` option can be used to specify the order to store bit field members in a field.

### d. Function Call in Extended Memory Indirect Addressing Mode (only in H8SX)

The `__indirect_ex` keyword or the `indirect=extended` option can be used to declare functions to be called in extended memory indirect addressing mode. Also, `#pragma indirect` section can modify the section name of not only `$INDIRECT`, the function address area for memory indirect addressing mode (`@@aa:8`), but also `$EXINDIRECT`, the function address area for extended memory indirect addressing mode (`@@aa:7`).

### e. Assembly Capability (only in H8SX)

The `__asm` keyword can be used to allow the assembly language to be used in a C/C++ source program.

### f. Disabling #line Output

The `noline` option can be used to disable the `#line` output at preprocessor expansion.

### g. Specifying Inline Expansion for Functions `memcpy` and `strcpy` (only in H8SX)

The `library` option can be used to specify inline expansion of two library functions, `memcpy` and `strcpy`.

### h. Changing Error Level

The **change\_message** option can be used to individually change the error level of information-level and warning-level error messages.

**i. Specifying 8-bit Absolute Area Address (only in H8SX)**

Option **sbr** can be used to specify the address to locate the 8-bit absolute area.

**j. Strengthening Optimizing Feature (only in H8SX)**

The optimization details can be further specified by the following added options:

`opt_range`, `del_vacant_loop`, `max_unroll`, `infinite_loop`, `global_alloc`, `struct_alloc`, `const_var_propagate`, and `volatile_loop`

**k. Added Intrinsic Functions**

The following intrinsic functions are added.

- 64-bit multiplication of H8SX (`mulsu` and `muluu`)
- Block transfer instructions of H8SX (`movmdb`, `movmdw`, `movmdl`, and `movsd`)
- Block transfer instructions (`eepmovb`, `eepmovw`, `eepmovi`)
- Revised intrinsic function for MOVFPE instruction (`_movfpe`)

**l. Support for Wild Cards**

An input file can be specified with a wild card.

**m. Change in Compiler Limitation**

The limitation in the number of **switch** statements is changed from 256 to 2048.

**n. Change in specification of information message display**

In Ver. 4.0, only the last specification of all the **message** and **nomessage** options was effective in a command line. In Ver. 6.0, the union of all the numbers specified by each **nomessage** option in a command line is suppressed to display the message.

**o. Type of enum instance**

If the **byteenum** option is specified and if all the numbers in an enum are in the range from 0 to 255, the compiler handles the data as **unsigned char**.

**p. Inline expansion**

In H8SX, `<numeric value>` in the **speed=inline=<numeric value>** option means the percentage of increase in program size allowed by inline expansion. In the other CPU, `<numeric value>` means the maximum number of nodes in a function allowed to perform inline expansion.

**q. 1-byte-aligned Data Section and 4-byte-aligned Data Section (only in H8SX)**

Specifying the `align=4` option places data whose size is odd to 1-byte-aligned data section and data whose size is a multiple of 4 to 4byte-aligned data section.

#### r. Section Name

Changing the section name of P, C, B or D into S by the **section** option causes a warning error. S is the reserved name for the stack area.

#### s. Added Implicit Declarations

`__H8SXN__`, `__H8SXM__`, `__H8SXA__`, `__H8SXX__`,  
`__HAS_MULTIPLIER__`, `__HAS_DIVIDER__`, `__INTRINSIC_LIB__`,  
`__DATA_ADDRESS_SIZE__`, `__H8__`, `__RENESAS_VERSION__`, and  
`__RENESAS__` are implicitly declared using `#define` directive by the compiler.

#### t. Reentrant library

If the **reent** option is specified to the library generator, a reentrant library is created.

#### u. Support of Little-endian Space (only in H8SX)

A little-endian space is supported depending on a chip of H8SX. A 2 -or 4--byte datum in a little-endian space should be written and read with its own data size. In order to do so, the feature of the `__evenaccess` keyword is enhanced.

### 3. Added and Improved Features in Upgrade from Ver. 6.0 to Ver. 6.1

#### a. Support for AE5

AE5 is supported.

#### b. Enhanced Conformance with the ANSI Standard

**strict\_ansi** brings the associative rule of floating-point operations into conformance with the ANSI standard.

#### c. Compatibility of Output Object Code with Object Code Produced by Ver. 4.0

With H8S CPUs, **legacy=v4** supports the output of object code which is compatible with that produced by earlier versions of the compiler (Ver.4.0).

#### d. Expanded Specifications of **cpuexpand=v6** Specified with **legacy=v4**

When **cpuexpand=v6** is specified with **legacy=v4**, output object code is compatible with object code produced by Ver. 6.00 and the **cpuexpand** option.

#### e. Preferential Allocation of Register Storage Class Variables

**enable\_register** preferentially allocates the variables with register storage class specification to registers.

#### f. Division of Optimizing Ranges

**scope/noscope** can be specified to select whether or not to divide up ranges for optimization within functions.

**g. Inter-file Inline Expansion**

**file\_inline** is used to specify inline expansion for functions that extend across files and **file\_inline\_path** is used to specify the path name of a file for inline expansion.

**h. Added Intrinsic Function**

Intrinsic function **set\_vbr** is used to set the VBR.

**i. #pragma address**

**#pragma address** can be used to allocate variables to specific absolute addresses.

**j. Support for .stack Directive**

When **code=asmcode** has been specified, the compiler outputs a .stack directive within the assembly-source program.

**k. Added Environment Variable**

Environment variable **CH38SBR** can be used to set initial values for the SBR.

**l. Added Implicit Declarations**

Implicit declaration of `__AE5__` and `__ABS16__` have been added.

**4. Notes on Optimizing Features of the Compiler Ver. 6.01**

Notes below about optimization apply in a case where an H8SX and H8S (without the **legacy=v4** option) object program is created with Ver. 6.01 optimization. For the other cases, optimization is similar to that of Ver. 4.0 or earlier.

Adopting the newest compiler optimization technology allows the optimization processing in Ver. 6.01 to analyze aliases for pointers or external variables and analyze data live ranges including the control flow, which were not possible so far (in Ver. 4.0 or earlier). This provides a wider range of optimization than Ver. 4.0 within the limits of the language specifications.

However, a program that was previously running because it was not optimized enough may not run because it has become a target of optimization.

Examples of programs that were not optimized so far but will become targets of optimization in Ver. 6.01 are shown below.

**a. Access to External Variables or Pointer Variables without volatile Declaration**

A **volatile** declaration guarantees that the volatile-qualified variable is accessed whenever it is used because the variable may be updated outside the program sequence. For example, data values are changed by interrupt processing or hardware processing.

The compiler assumes that variables without a **volatile** declaration are changed only by successive processing of the program sequence or function calls.

In Ver. 4.0 or earlier, external variables without a **volatile** declaration were optimized as shown in the example below:



Example:

```
int a;
f() {
    int *ptr=&a;
    *ptr=1;           //<- Only this assignment expression is eliminated.
    *ptr=2;
}
```

In Ver. 6.01, optimization is further performed in the cases below.

To disable the optimization, declare the relevant variable with **volatile**.

Example 1:

```
int a;
f() {
    int *ptr=&a;
    *ptr &= ~( 0x0080 ) ;           //<- (1)
    while( !( *ptr & (0x0080) ) )  //<- (2)
    }
    :
}
```

In this example, **while** statement (2) has become an infinite loop as a result of optimization.

- Due to alias analysis of the pointer, \*ptr in (1) and \*ptr in (2) are handled as the same value.
- Expression (1) is propagated to expression (2). Accordingly, expression (2) is converted as follows:

```
while( !( (*ptr & ~( 0x0080 ) ) & (0x0080) ) ) //<- (2)
-> while(!( *ptr & 0 ))
-> while(!(0))
-> while(1)
```

Therefore, the expression in question is judged as true, the judge statement is eliminated, and the above **while** statement becomes an infinite loop.

### Example 2:

```
int a,b;
f() {
    a=1;           //<- (1)
    if(a);         //<- (2)
    {
        b=1;       //<- (3)
    }
}
```

In this example, **if** statement (2) has been eliminated and (3) is always executed at all times as a result of optimization.

- Due to alias analysis of external variables, a in (1) and a in (2) are handled as the same value.
- Constant value (1) is propagated to expression (2). Accordingly, expression (2) is converted as follows:

```
-> if(1)
```

Therefore, the expression in question is judged as true, the conditional statement is eliminated, and the above expression (3) is always executed at all times.

### Example 3:

```
int a,b,c;
f() {
    a=1;           //<- (1)
    if(c);         //<- (2)
    {
        b=1;       //<- (3)
    }
    a=2;           //<- (4)
}
```

In this example, expression (1) has been eliminated as a result of optimization.

- Obtains the control flow including the conditional of the if statement expression.
- Due to analyzing the control flow analysis and alias analysis of external variables, it is proved that the value set in a in (1) is not used. Therefore, the above expression (1) is a redundant expression that is not referenced, and thus it is eliminated.

#### Example 4:

```
int a;
int b[10];
f() {
    int i;                //<- (1)
    for(i=0; i<10; i++)    //<- (2)
    {
        b[i]=a;           //<- (3)
    }
}
```

In this example, a in expression (3) is referenced once before the loop and is always handled as a constant value in the loop as a result of optimization.

- Obtains the control flow including the **for** loop control expression.
- Due to analyzing the control flow analysis and alias analysis of external variables, a in (3) is handled as a constant value in the loop.
- (3) which is the reference expression to a is moved outside the **for** loop (2) as follows:

```
temp=a;
for(i=0; i<10; i++)    //<- (2)
{
    b[i]=temp;         //<- (3)
}
```

Therefore, the variable a in expression (3) is unchanged in the loop.

#### Example 5:

```
int a;
f() {
    a=0;                //<- (1)
    while(1);           //<- (2)
}
```

In this example, the statement (1) is assumed as unnecessary and eliminated as a result of optimization.

- Since (2) is an infinite loop, this function is judged to have no exit.
- Since a is not referenced in the infinite loop, specification (1) is assumed as unnecessary coding and is eliminated.

## b. **volatile\_loop** Option

If the loop control variable is a non-volatile external variable and also the conditional expression is simple, the **volatile\_loop** option regards the loop control variable as **volatile**-qualified to prevent an infinite loop from being created. However, if the loop control variable is not loop-invariant, it cannot be treated as **volatile**-qualified.

In Ver.6.01, declare the relevant variable with **volatile**.

An example program is given below.

Example:

```
struct{
    unsigned char a:1;
} ST;
int a;
extern void f();
void func() {
    while (ST.a) {           //<- (1)
        if (a) {             //<- (2)
            f();              //<- (3)
        }
    }
}
```

In this example, because ST.a may be updated in f(), ST.a is not assumed as loop-invariant value in the loop. Therefore, ST cannot be treated as **volatile** even though specified so with the **volatile\_loop** option.

- If the condition in (2) is satisfied, (3) is executed and the ST.a value may be updated. Accordingly, after the function call, ST.a is to be reloaded.
- If the condition in (2) is not satisfied, the ST.a value is not updated so the ST.a value used in the previous conditional at (1) can be directly used.

## 5. Compatibility between Ver. 4.0 and Ver. 6.01

To link an object program created by Ver. 4.0 with an object program created by Ver. 6.01, the following conditions need to be satisfied.

### (1) C source program

The following options that affect function interface must be specified equally.

- regparam
- longreg/nolongreg
- double=float
- structreg/nostructreg
- stack
- byteenum
- pack/unpack

### (2) Assembly program

An assembly program must conform to the rules concerning function call, which are described in section 9.3.2, Function Calling Interface.

- Notes:
1. For information not mentioned in the manual, the compatibility with an upgraded version is not guaranteed. An object program created by Ver. 4.0 cannot be linked with an object program created by Ver. 6.01 if one or both of the object programs contain assembly coding which depends on the compiler output coding, such as the order to save and restore register contents.
  2. For details on linkage with an OS, middleware, and so on, contact your sales agency.

## 18.2.3 Added and Improved Features for the Assembler

### 1. Added and Improved Features in Ver. 4.0

#### a. External Definition and Reference of BEQU

The .BEQU symbol can be externally defined and referenced by using .BIMPORT and .BEXPORT.

### 2. Added and Improved Features in Upgrade from Ver. 4.0 to Ver. 6.0

(Note: Ver. 5.0 does not exist and is a missing number.)

#### a. Support for the New CPU

Creation of an object file with a CPU type of H8SX is supported.

#### b. Adding Check on Use of a Register

The following warning will be detected if @Rn+, @-Rn, @+Rn, @Rn-, @(d,Rn) or @Rn is described on the program with H8SX, H8S, or H8/300H CPU.

819 (W) @Rn+, @-Rn, @+Rn, @Rn-, @(d,Rn) OR @Rn USED

Change Rn into ERn in the above addressing modes.

### **3. Added and Improved Features in Upgrade from Ver. 6.0 to Ver. 6.01**

#### **a. Support for the New CPU**

AE5 is supported.

#### **b. Loosening Limits on Values**

The limitation on the number of characters in a replacement symbol specified in the **DEFINE** option or directive is loosened from 32 characters to unlimited. (However, the replacement string literal is still limited to 255 characters.)

#### **c. Exemptions from Replacement by the DEFINE Option or Directive**

DEFINE options and directives do not replace **.AENDI**, **.AENDR**, **.AENDW**, **.AIFDEF**, **.END**, **.ENDF**, **.ENDM**, **.ENDI**, **.ENDS**, and **.ENDW** directives.

#### **d. Support for .STACK Directive**

The **.STACK** directive enables specification of a stack size for use with a specific symbol.

### **18.2.4 Added and Improved Features for the Optimizing Linkage Editor**

#### **1. Added and Improved Features in Ver. 7.0 and Ver. 7.1**

##### **a. Support for Wild Cards**

A wild card can be specified with a section name of an input file or for file names in start options.

##### **b. Search Path**

An environment variable (HLNK\_DIR) can be used to specify several search paths for input files or library files.

##### **c. Subdividing the Output of Load Modules**

The output of an absolute load module file can be subdivided.

##### **d. Changing the Error Level**

For information, warning, and error level messages, the error level or disabling/enabling the output can be individually changed.

##### **e. Support for Binary and HEX**

Binary files can be input and output.

Intel® HEX-type output can be selected.

##### **f. Output of the Stack Consumption Information**

The stack option can output an information file for the stack consumption analysis tool.

**g. Optimization Improvement by `optimize=variable_access`**

Variables allocated in a 16-bit absolute address space can be allocated in an 8-bit address space by applying optimization.

**h. Optimization Improvement by `optimize=register`**

When option **`optimize=speed`** is not specified, the file is compressed after optimizing the saving and restoring of register contents between functions, and replacing saving and restoring of multiple register contents with function calls.

**i. Optimization Improvement of Assembly Programs**

Sections including `.org`, `.align`, or `.data` directive can be optimized.

**j. Debugging Information Deletion**

The **`strip`** option can be used to delete debugging information from either the load module file or the library file.

**k. Debugging Information Compression**

The **`compress`** option can be used to compress debugging information.

**2. Added and Improved Functions in Upgrade from Ver. 7.0 to Ver. 8.0**

**a. Support for New CPU**

Input of an object file with a CPU type of H8SX is supported.

**b. Output to Empty Area**

The **`space`** option can be used to fill the specified value in an empty area.

**c. Specifying Memory Size Used**

The **`memory`** option specifies the used size of internal memory.

**d. Specifying 8-bit Absolute Area Address**

The **`sbr`** option specifies the address to locate the 8-bit absolute address area.

**e. Changing Error Level for Overlapping Section Addresses**

The error level for overlapping section addresses at linkage is changed from Fatal in Ver. 7.0 to Error in Ver. 8.0. Thus, even when the section addresses overlap, the **`change_message`** option can be used to continue processing on the user's own responsibility.

### 3. Added and Improved Functions in Upgrade from Ver.8.00 to Ver. 9.00

#### a. Alignment Value Specification for Input Section with binary Option

A boundary alignment value can be specified for the section specified by the **binary** option.

#### b. Output of Cross-Reference Information

The cross-reference information is output to the linkage list when the **show=xreference** option is specified, which is useful to determine the location that refers to a variable or function.

#### c. Notification of Unreferenced Symbol

When the **msg\_unused** option is specified, the user can be notified of unreferenced symbols even if optimization is not specified.

#### d. Reducing Empty Areas between Sections

In compiler units, the **data\_stuff** option tightens up the spacing between areas in the compiler output.

## 18.3 Operating Format Converter

### 18.3.1 Object File Format

The object file format complies with the standard ELF format. The debugging information format complies with the standard DWARF2 format.

### 18.3.2 Compatibility with Earlier Versions

#### 1. Object and Library Files

When an object file or library file that has been output by Ver. 3.0 or earlier version of the compiler or assembler is to be input to the optimizing linkage editor, it must be converted to the ELF format by using a format converter. However, the debugging information will then be deleted.

Relocatable files that have been output by the linkage editor (extension: rel) and library files that include such relocatable files cannot be converted.

The format converter outputs a file with a converted object format and the same name as the input file. The input file is saved as <input file name.extension>.bak.

ELF-format object and library files cannot be converted to the object format of earlier versions of the compiler (Ver. 3.0 or earlier) or assembler.

#### 2. Load Module File

ELF-format load module files can be converted to the format of the linkage editor of Ver. 6.0 or earlier versions by using the format converter. Table 18.3 is a list of the object file formats that can be converted. The ELF-format load module file of H8SX is not supported.



**Table 18.3 Object File Formats that can be Converted from ELF Format**

No.	Version Number of Compiler or Assembler	Object File Format			
		Linkage Editor Specification Option		Object	Debugging Information
1	Ver. 2.0 or lower	debug		SYSROF	SYSROF
2		sdebug		SYSROF	SYSROF
3	Ver. 3.0 or lower	sysrof	debug	SYSROF	SYSROF
4			sdebug	SYSROF	DWARF1
5		elf	debug	ELF	DWARF1
6			sdebug	ELF	DWARF1

The format converter outputs a converted file with the same file name as the input file. The input file is saved as <input file name.extension>.bak.

The load module file output by the linkage editor of Ver. 6.0 or earlier versions cannot be converted to the ELF format.

Load modules with a newly-added feature of the compiler, assembler, or optimizing linkage editor in the Ver. 4.0 package or later cannot be converted from the ELF format to one of the older formats.

### 18.3.3 Command Line Format

The command line format is as follows:

```
helfcnv[Δ<option>...]Δ<file name>[...][Δ<option>...]
```

<option>: -<option>[=<suboption>]

<file name>: A wild card (\* or ?) can be used.

### 18.3.4 Interpretation of Options

In the command line format, uppercase letters indicate the abbreviations and characters underlined indicate the defaults. When the HEW is used, the option is specified in the option window of the optimizing linkage editor. The format of the dialog menus of the HEW is

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

The format converter automatically determines the type of the file to be converted (object file, library file, or load module).

#### 1. Conversion of Object File or Library File

An object file or library file created by Ver. 3.x or earlier versions of the compiler or assembler is converted to the ELF format. The debugging information that was included in the object file or library file is deleted.

Use this function from the command line since it is not supported by the HEW.

**Table 18.4 Options for Converting Object Files or Library Files**

No.	Item	Option	Dialog Menu	Specification
1	Address space specification	Address_space=<size> <size>:{ 20   <u>24</u>   28   32 }	—	Address space specification
2	fpu	Fpu	—	With FPU*
3	dsp	Dsp	—	With DSP*

Note: Options for the SuperH compiler. They cannot be used in the H8S, H8/300 series compiler.

### Address\_space: Address Space Specification

- Command Line Format

Address\_space=<Address space size>

<Address space size>:{ 20 | 24 | 28 | 32 }

- Description

Specifies the address space size when **cpu=300ha**, **2000a**, or **2600a** is specified.

When the option is omitted, **24** will be selected.

- Example

```
helfcnv -a=20 *.obj ; Converts all files with extension .obj to files with
; extension .elf and a 20-bit address space.
```

- Remarks

Library files that include relocatable files output by the linkage editor (extension: .rel), and other relocatable files cannot be converted.

## 2. Conversion of Load Module Files

ELF-format load module files have been converted to the object file format output by of the linkage editor of Ver. 6.0 or earlier versions. When debugging information is included in the load module file, the load module file after conversion retains the debugging information.

Refrain from converting a load module of H8SX.

**Table 18.5 Options for Converting Load Module File**

	Item	Option	Dialog Menu	Specification
1	Specification of conversion format	<u>Sysrof</u>	Output	Converts to the SYSROF format
		Dwarf1	[Type of output file:]	Converts to the ELF/DWARF1 format

## Sysrof, Dwarf1: Conversion Format Specification

Link/Library <Output> [Type of output file:][Absolute(SYSROF)]

- Command Line Format

Sysrof

Dwarf1

- Description

Specifies the object format after conversion.

When **sysrof** is specified, a load module file in the ELF/DWARF2 format is converted to the SYSROF format.

When **dwarf1** is specified, a load module file in the ELF/DWARF2 format is converted to the ELF/DWARF1 format.

When the **sdebug** option is specified to the optimizing linkage editor, the debugging information is not retained in the converted file.

- Example

`helfcnv test.abs ; Converts test.abs to the SYSROF format.`

`helfcnv -d test.abs ; Converts test.abs to the ELF/DWARF1 format.`

- Remarks

When there are functions both with and without optimization in a file because **#pragma** option has been used, debugging information will not be included in the converted file.

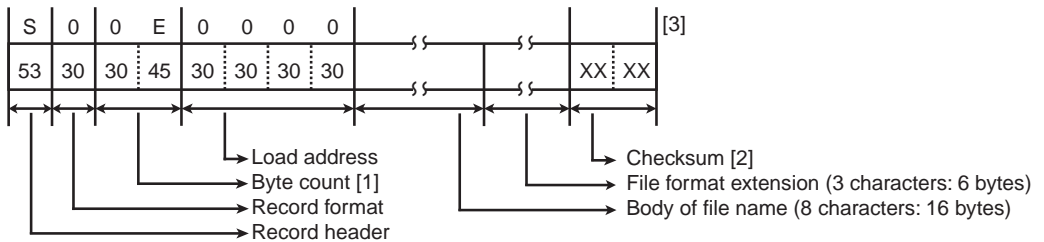


## 19.1 S-Type and HEX File Format

This section describes the S-type files and HEX files that are output by the optimizing linkage editor.

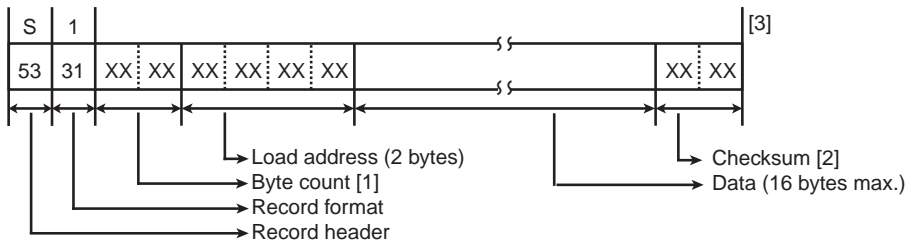
### 19.1.1 S-Type File Format

#### (a) Header record (S0 record)

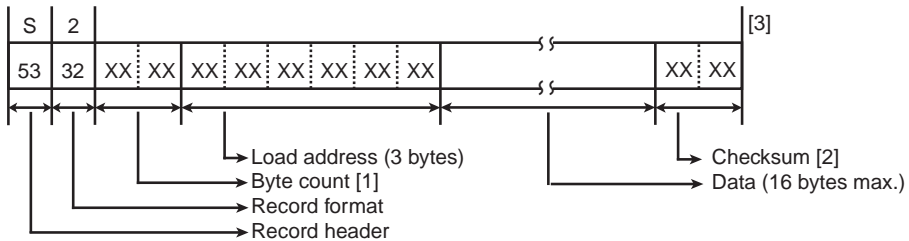


#### (b) Data record (S1, S2, and S3 records)

##### (i) When the load address is 0 to FFFF

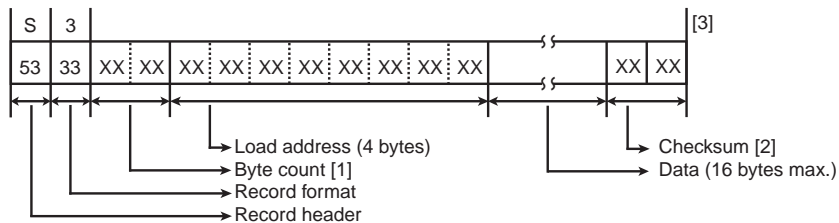


##### (ii) When the load address is 10000 to FFFFFF



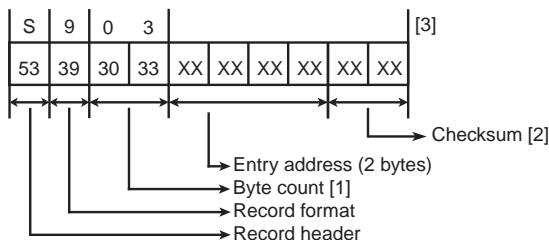
**Figure 19.1 S-Type File Format**

(iii) When the load address is 1000000 to FFFFFFFF

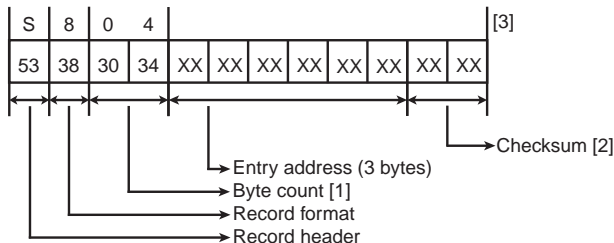


(c) End record (S9, S8, and S7 record)

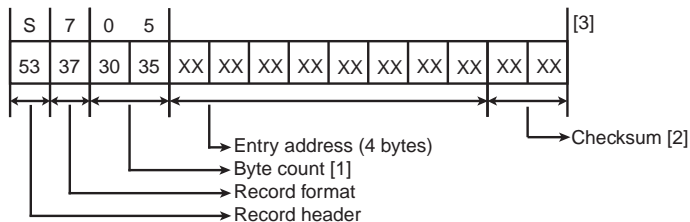
(i) When the entry address is 0 to FFFF



(ii) When the entry address is 10000 to FFFFFF



(iii) When the entry address is 1000000 to FFFFFFFF



- Notes:
- [1] The number of bytes from the load address (or the entry address) to the checksum.
  - [2] 1's complement of the sum of the byte count and the data between the checksum and the byte count, in byte units.
  - [3] A new-line character is added immediately after the checksum.

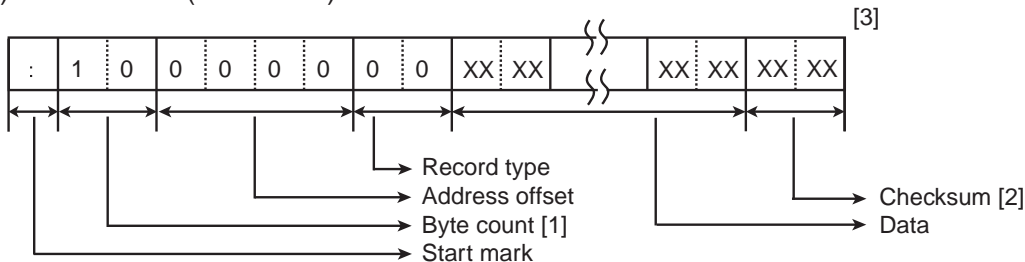
**Figure 19.1 S-Type File Format (cont)**

## 19.1.2 HEX File Format

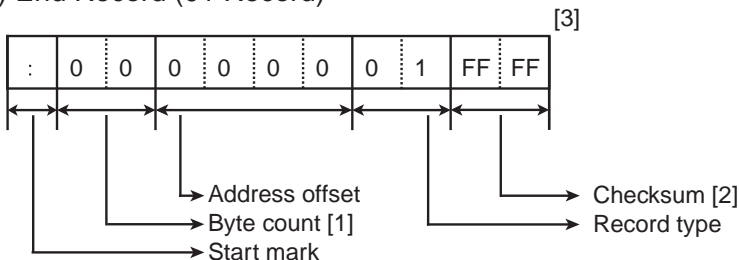
The execution address of each data record is obtained as described below.

- Segment address  
(Segment base address  $\ll 4$ ) + (Address offset of the data record)
- Linear address  
(Linear base address  $\ll 16$ ) + (Address offset of the data record)

(a) Data Record (00 Record)



(b) End Record (01 Record)



(c) Expansion Segment Address Record (02 Record)

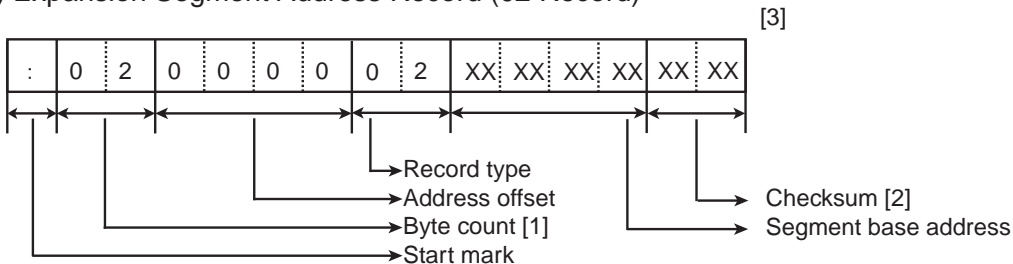


Figure 19.2 HEX File Format

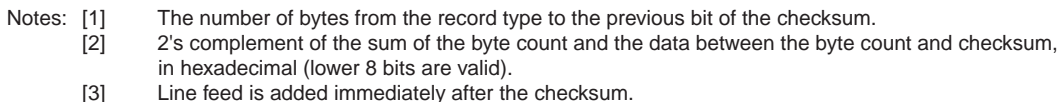
[3]



[3]



[3]



### Figure 19.2      HEX File Format (cont)



## 19.2 ASCII Code List

Table 19.1 ASCII Code List

Lower 4 bits	Upper 4 bits							
	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	–	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

## 19.3 Access Range of Short Absolute Addresses

Table 19.2 shows the access range of 8-bit absolute addresses and 16-bit absolute addresses in CPU/operating mode.

**Table 19.2 Access Range of Short Absolute Addresses**

<b>CPU/Operating Mode</b>	<b>Access Range of 8-Bit Absolute Addresses (@aa:8)</b>	<b>Access Range of 16-Bit Absolute Addresses (@aa:16)</b>
H8SXA:32 H8SXX[:32] 2600a:32 2000a:32	0xFFFFF00 to 0xFFFFFFFF	0x0 to 0x7FFF 0xFFFF8000 to 0xFFFFFFFF
H8SXA:28 H8SXX:28 2600a:28 2000a:28	0xFFFFF00 to 0xFFFFFFFF	0x0 to 0x7FFF 0xFFFF8000 to 0xFFFFFFFF
H8SXA[:24] H8SXM[:24] 2600a[:24] 2000a[:24] 300ha[:24]	0xFFFF00 to 0xFFFFF	0x0 to 0x7FFF 0xFF8000 to 0xFFFFF
H8SXA:20 H8SXM:20 2600a:20 2000a:20 300ha:20	0xFFF00 to 0xFFFFF	0x0 to 0x7FFF 0xF8000 to 0xFFFFF
H8SXN 2600n 2000n 300hn 300 300l	0xFF00 to 0xFFFF	—

Note: When the H8SX is selected as the CPU, the access range of 8-bit absolute addresses can be modified by the **sbr** option.

# Index

## #

#pragma abs16.....	317
#pragma abs16 section.....	314
#pragma abs8.....	317
#pragma abs8 section.....	314
#pragma address .....	353
#pragma asm.....	341
#pragma bit_order.....	301, 322
#pragma entry .....	328
#pragma extension .....	311
#pragma global_register .....	347
#pragma indirect .....	331
#pragma indirect section.....	314
#pragma inline .....	334
#pragma inline_asm.....	335
#pragma interrupt.....	323
#pragma noregsave .....	337
#pragma option .....	339
#pragma pack 1.....	348
#pragma pack 2.....	348
#pragma regsave .....	337
#pragma section .....	314
#pragma stacksize.....	314
#pragma unpack.....	348

## \$

\$ .....	637
\$\$function_name .....	333
\$1 .....	187
\$4 .....	187
\$ABS16B.....	32, 186, 187, 198, 315, 317
\$ABS16C.....	32, 186, 187, 198, 315, 317
\$ABS16D.....	32, 186, 187, 198, 315, 317
\$ABS8B.....	32, 186, 198, 315, 317
\$ABS8C.....	32, 186, 198, 315, 317
\$ABS8D.....	32, 186, 198, 315, 317
\$ADDRESS .....	188
\$EXINDIRECT .....	31, 187, 198, 315
\$function_name .....	331

\$INDIRECT.....	31, 187, 198, 315, 331
\$VECT.....	187
* .....	
* specification.....	472
.	
.ABS8 .....	723
.AELIF.....	764
.AELSE.....	764, 766
.AENDI.....	764, 766
.AENDR .....	768
.AENDW .....	769
.AERROR.....	772
.AIF.....	764
.AIFDEF .....	766
.ALIGN.....	689
.ALIMIT .....	773
.AREPEAT .....	768
.ASSIGN.....	692
.ASSIGNA .....	759
.ASSIGNC .....	761
.AWHILE .....	769
.BEQU .....	695
.BEXPORT .....	721
.BIMPORT .....	722
.BREAK.....	797, 812
.CASE.....	797
.CONTINUE.....	813
.CPU .....	676
.DATA .....	697
.DATAB .....	699
.DEBUG .....	727
.DEFINE.....	763
.DISPSIZE .....	730
.ELSE.....	793
.END .....	747
.ENDF.....	802
.ENDI.....	793
.ENDM .....	777
.ENDS.....	797
.ENDW .....	807
.EQU .....	691

.EXITM.....	771
.EXPORT.....	715
.FOR[U].....	802
.FORM.....	737
.GLOBAL.....	719
.HEADING.....	739
.IF.....	793
.IMPORT.....	717
.INCLUDE.....	750
.INSTR.....	788
.LEN.....	787
.LINE.....	729
.LIST.....	735
.MACRO.....	777
.NOABS8.....	723
.ORG.....	686
.OTHERS.....	797
.OUTPUT.....	725
.PAGE.....	741
.PRINT.....	733
.PROGRAM.....	745
.RADIX.....	746
.REG.....	693
.REPEAT.....	810
.RES.....	708
.SDATA.....	701
.SDATAB.....	702
.SDATAAC.....	704
.SDATAZ.....	706
.SECTION.....	681
.SPACE.....	743
.SRES.....	710
.SRESC.....	711
.SRESZ.....	713
.STACK.....	748
.SUBSTR.....	789
.SWITCH.....	797
.UNTIL.....	810
.WHILE.....	807
?	
? .....	649

__abs16.....	33, 317
__abs8.....	33, 317
__asm.....	342
__entry.....	328
__evenaccess.....	351
__global_register.....	347
__indirect.....	331
__indirect_ex.....	333
__inline.....	334
__interrupt.....	323
__near16.....	319
__near8.....	319
__noregsave.....	337
__ptr16.....	321
__regparam2.....	338
__regparam3.....	338
__regsave.....	337
__secend.....	359
__sectop.....	359
__CALL_END.....	206
__CALL_INIT.....	206, 212
__CLOSEALL.....	206, 214
__ec2p_new_handler.....	576
__file_ptr.....	574
__INITLIB.....	206, 212, 215
__INITSCT.....	206
__IOFBF.....	453
__IOLBF.....	453
__IONBF.....	453
__movfpe.....	377

+	
+	630

## 2

2000A.....	53, 88, 676
2000N.....	53, 88, 676
2600A.....	53, 88, 676
2600N.....	53, 88, 676

## 3

300.....	53, 88, 676
----------	-------------

300HA .....	53, 88, 676
300HN .....	53, 88, 676
300L.....	53, 88, 676
300REG .....	53
<b>6</b>	
64-bit multiplication .....	362, 374
<b>8</b>	
8-bit absolute area address specification.....	59
<b>A</b>	
abort.....	94, 243
abs.....	509, 586, 596
abs16.....	32, 84
abs8.....	32, 84
absolute.....	109, 185
absolute_forbid .....	123
access to EEPROM.....	931
access to EEPROM.....	926
acos .....	417
acosf.....	432
add .....	940
address check.....	127
address symbol.....	631
address_space .....	958
addressing mode .....	650
AE5.....	88, 925
aliases of register name.....	631
align .....	19
ALIGN.....	681
align_section .....	939
alignment .....	19, 348
all .....	18
allocation .....	23, 24
allocation of initialized data areas.....	198
alphabetic character .....	401
and_ccr.....	367
and_exr .....	370
ANSI conformance .....	49
arg.....	587, 597
argument reference in macro .....	780
array .....	283

array type .....	293
asin .....	418
asinf .....	432
asmcode .....	14
assembler directives .....	674
assembler functions .....	930
assembler listings .....	167
assembly program sections .....	189
assembly specifications .....	627
assembly-language instruction .....	341
assert .....	399
assert.h .....	399
assigna .....	69
assignc .....	70
atan .....	418
atan2 .....	419
atan2f .....	434
atanf .....	433
atof .....	500
atoi .....	501
atol .....	501
auto .....	18, 29

## **B**

B .....	186, 198, 315
basic type .....	286
binary .....	104, 109
binary file .....	104, 394
binary number .....	635
bit data name .....	631
bit field .....	285, 299, 351
bit field order specification .....	60
bit_order .....	60, 301
block transfer .....	379, 380
block transfer instruction .....	47
block transfer instructions (with ECR setting) .....	928
block transfer instructions (with EPR and ECR setting) .....	929
bool .....	289, 555, 565
boolalpha .....	544
boundary alignment .....	185, 288
boundary alignment of structure, union, and class members .....	58
boundary alignment value and disable of boundary alignment .....	19
br_relative .....	76



branch .....	121
bsearch .....	507
bss .....	15
BTBL .....	205, 210
BUFSIZ .....	453
byteenum .....	45

## C

C .....	61, 186, 198, 315
C library function.....	390
C\$BSEC.....	188, 198
C\$DSEC .....	188, 198
C\$INIT.....	188, 198
C\$VTBL .....	188, 198
C/C++ language specifications .....	279
C/C++ libraries .....	390
C/C++ library function initial settings .....	206, 212
C/C++ program sections .....	185
C++ class libraries .....	533
C++ global class object initial settings .....	212
cache size .....	123
cachesize .....	123
calculation of heap area size .....	203
calculation of size .....	196
calculation of stack area size .....	200
callee-save .....	247
caller-save.....	247
calloc.....	505
calls .....	81
case .....	29
ceil .....	427
ceilf .....	442
CH38.....	151
CH38SBR .....	151
CH38TMP .....	151
change_message .....	42, 48, 135, 145
char .....	289
CHAR_BIT.....	413
CHAR_MAX.....	413
CHAR_MIN .....	413
character code select in string literal.....	62
character constant .....	635
character specifications.....	280

class .....	285
class data allocation .....	295
class type.....	293
clearerr.....	495
close .....	218
close files .....	206
closing files.....	214
cmncode.....	47
code.....	14, 81, 185
CODE .....	681
coding notes.....	272
columns.....	97
command line format .....	7, 65, 101, 141, 957
command-line interface .....	937
comment .....	43, 628
comment in macro .....	783
comment nesting.....	43
common code size.....	121
common expression optimization .....	47
compatibility with earlier versions.....	956
compatibility with the earlier version .....	934
compiler functions .....	925
compiler listings.....	157
compiler options .....	925
compiler specifications .....	279
compiling a C Program with the C++ Compiler .....	275
complex .....	533, 578
complex number calculation class libraries .....	578
compound type.....	293
compress .....	131
condition code register.....	361
conditional assembly .....	754
conditional assembly directives.....	758
conditional assembly function .....	752
conditional iterated expansion .....	757
conditionals.....	81
conj .....	587, 597
const.....	15
const constant propagation.....	39
const_var_propagate .....	39
constant.....	635
constant areas.....	198
constant symbol .....	631

contents of dynamic memory area .....	199
contents of static memory area .....	196
control character .....	401
conversion between decimal and internal representation.....	308
conversion character .....	456
conversion specification .....	465, 472
conversion specifier .....	472
converted data size.....	472
copyright.....	139
cos.....	420, 587, 597
cosf .....	435
cosh.....	421, 587, 597
coshf .....	436
cpp .....	61
cpu .....	926
cpu .....	930
cpu .....	52, 88, 127
CPU option .....	138
CPU options.....	50, 87
cpu type.....	926
cpu type.....	930
CPU type.....	925
CPU/operating mode .....	52
cpuexpand.....	16
cross reference listing .....	169
cross_reference .....	82
cross-reference information .....	179
ctype.h.....	400

## **D**

D .....	52, 90, 186, 198, 315, 678
dadd .....	388
data .....	15, 185
DATA .....	681
data allocation example .....	288
data range.....	288
data representation .....	288
data_stuff .....	115
DBL_DIG .....	411
DBL_EXP_DIG.....	411
DBL_MANT_DIG .....	411
DBL_MAX.....	409
DBL_MAX_10_EXP .....	410

DBL_MAX_EXP .....	410
DBL_MIN .....	410
DBL_MIN_10_EXP .....	411
DBL_MIN_EXP .....	410
DBL_NEG_EPS .....	412
DBL_NEG_EPS_EXP .....	412
DBL_POS_EPS .....	412
DBL_POS_EPS_EXP .....	412
debug .....	14, 72, 111
debug information .....	14, 111
debug information compression .....	131
debug information deletion .....	134
dec .....	546
decimal digit .....	401
decimal number .....	635
decimal operation .....	361
declaration specifications .....	286
default include file .....	9
define .....	10, 68, 104
definitions .....	81
del_vacant_loop .....	36
delete .....	132
denormalized number .....	303, 304, 306
diagnostics .....	399
disable of copyright output .....	62
disable preprocessor inline output .....	145
disabling optimization against loop iteration condition .....	45
div .....	509
div_t .....	499
divider .....	678
division of optimizing ranges .....	40
domain error .....	416, 430
double .....	289, 302, 305
double to float conversion .....	55
double_complex class .....	589
double_complex non-member function .....	592
double_complex::_im .....	589
double_complex::_re .....	589
double_complex::double_complex .....	590
double_complex::imag .....	590
double_complex::operator-= .....	590, 591
double_complex::operator*= .....	591
double_complex::operator/= .....	591

double_complex::operator+=.....	590, 591
double_complex::operator= .....	590, 591
double_complex::real .....	590
double_complex::value_type .....	589
double=float.....	55
dsub.....	389
DTBL.....	205, 210
DUMMY .....	681
dwarf1 .....	959
dynamic memory area allocation .....	199, 204

## ***E***

EBADF .....	414, 884
ECBASE.....	414, 883
ecpp.....	44
ECSPEC.....	414, 884
EDBLO .....	414
EDBLU .....	414
EDIV .....	414, 883
EDOM .....	414, 415, 416, 429, 883
eepmov.....	47, 379
eepmovb.....	379
eepmovi .....	380
eepmovw.....	379
EEPROM .....	926
EEPROM .....	931
eepromb .....	928
eepromb_epr .....	929
eepromw .....	928
eepromw_epr .....	929
EEXP .....	414, 883
EEXPN .....	414, 883
EFLOATO .....	414, 884
EFLOATU .....	414, 884
ELDBLO .....	414
ELDBLU .....	414
elements of expression.....	638
elimination of expression preceding infinite loop.....	37
embedded C++ language .....	44
enable_register.....	49
end .....	140
end code.....	130
endl .....	570

end-of-file indicator .....	396
ends .....	570
entry .....	105
enum .....	289
enumeration .....	285
enumeration data size .....	45
environment .....	280
environment specifications .....	279
environment variables list .....	149
eof .....	538
EOF .....	393
EOVER .....	884
ERANGE .....	414, 415, 416, 429, 883
errno .....	397, 414
errno.h .....	414
errno_addr .....	220, 238
error .....	48, 135, 815, 885, 903, 917
error indicator .....	396
error information .....	160, 173, 182
error messages of assembler .....	885
error messages of C library function .....	882
error messages of compiler .....	815
error messages of optimizing linkage editor .....	903
error messages of standard library generator and format converter .....	917
ESTRN .....	414, 883
ETLN .....	414, 883
euc .....	62, 63, 95
EUNDER .....	884
exception .....	57
exception processing .....	57
exclude .....	86
executable instructions .....	650
execution continued .....	140
execution environment settings .....	205
execution start address .....	105
exit .....	140, 241
exp .....	423, 587, 597
expand .....	73
expanded memory indirect addressing mode .....	333
expansion .....	23
expansions .....	81
expf .....	438
exponent .....	302

expression .....	638, 641
expression .....	28
extend register.....	361
extended functions .....	311
external variable optimization .....	33
external variable optimization range specification .....	34
external variable register allocation .....	38
externally-defined symbol list .....	174
extract .....	134

## ***F***

fabs .....	427
fabsf .....	442
fatal .....	815, 885, 903, 917
fclose.....	459
feof.....	496
ferror .....	497
fflush.....	460
fgetc .....	481
fgets .....	482
field width.....	466, 467, 472
FILE.....	453
file access mode.....	395
file extension.....	155
file inclusion function .....	749
file pointer.....	393
file position indicator .....	396
FILE structure.....	393
file_inline.....	40
file_inline_path.....	11
fixed .....	546
flags .....	466
float.....	289, 302, 304
float.h.....	409
float_complex class .....	579
float_complex non-member function.....	582
float_complex::_im.....	579
float_complex::_re .....	579
float_complex::float_complex .....	580
float_complex::imag .....	580
float_complex::operator-= .....	580, 581
float_complex::operator*=.....	581
float_complex::operator/= .....	581

float_complex::operator+= .....	580, 581
float_complex::operator=.....	580, 581
float_complex::real .....	580
float_complex::value_type.....	579
floating-point number .....	395
floating-point number limits .....	282
floating-point number representation.....	303
floating-point number specifications .....	282, 302
floating-point numbers.....	391
floating-point operation specifications .....	307
floor .....	428
floorf.....	443
FLT_DIG .....	411
FLT_EXP_DIG.....	411
FLT_GUARD .....	409
FLT_MANT_DIG .....	411
FLT_MAX.....	409
FLT_MAX_10_EXP .....	410
FLT_MAX_EXP .....	410
FLT_MIN .....	410
FLT_MIN_10_EXP .....	411
FLT_MIN_EXP .....	410
FLT_NEG_EPS .....	412
FLT_NEG_EPS_EXP.....	412
FLT_NORMALIZE.....	409
FLT_POS_EPS .....	412
FLT_POS_EPS_EXP .....	412
FLT_RADIX .....	409
FLT_ROUNDS .....	409
flush .....	570
fmod.....	428
fmodf .....	443
fopen .....	461
form .....	108, 938
format converter.....	956
format type.....	185
fprintf.....	456, 465
fputc.....	483
fputs .....	484
fread .....	491
free .....	505
freopen.....	462
frexp.....	423



frexpf .....	438
fscanf .....	471
fseek .....	493
fsymbol .....	126
ftell .....	494
function access optimization symbol information .....	178
function address area .....	196, 198
function calling interface .....	246
function_call .....	121
function_forbid .....	123, 124
functions and macros .....	393
fwrite .....	492

## **G**

get_ccr .....	366
get_exr .....	370
get_imask_ccr .....	365
get_imask_exr .....	369
getc .....	485
getchar .....	486
getline .....	619
gets .....	487
global class object initialization processing .....	206
global class object postprocessing .....	206
global_alloc .....	38
goptimize .....	28, 77
guard bit .....	395

## **H**

H38CPU .....	150
H8/300 .....	670
H8/300H .....	666
H8/300L .....	670
H8S/2000 .....	662
H8S/2600 .....	658
H8SX .....	652
H8SXA .....	53, 88, 676
H8SXM .....	53, 88, 676
H8SXN .....	53, 88, 676
H8SXX .....	53, 88, 676
head .....	143
heap area .....	199
hex .....	546

hexadecimal .....	109
hexadecimal digit.....	401
hexadecimal number.....	635
hide .....	135
HIGH .....	644
HLNK_DIR .....	152
HLNK_LIBRARY1.....	152
HLNK_LIBRARY2.....	152
HLNK_LIBRARY3.....	152
HLNK_TMP .....	152
HUGE_VAL.....	415, 416, 429
HWORD .....	644

## ***I***

I/O.....	215
identifier specifications.....	279
ifthen.....	29
illegal operation .....	307
imag .....	586, 596
implementation definition.....	396
implicit declaration .....	153
improvements .....	941
include .....	9, 67
include file directory.....	9
increases the number of registers.....	46
increasing number of registers for register variables .....	46
indirect.....	31
indirect.h .....	332
infinite_loop.....	37
infinity .....	303, 305, 306, 457
information .....	48, 135, 815, 903
information message.....	10, 114
initial processing data area.....	198
initial setting program.....	195
initial settings.....	205, 208
initial value .....	185
initialized data areas .....	198
initialized data section address area.....	198
inline .....	28
inline expansion.....	39
input.....	3, 102
input file.....	102
input information .....	173

Input Information.....	181
input options .....	102
int.....	289
INT_MAX .....	413
INT_MIN.....	413
int_type .....	534
integer constant.....	635
integer specifications .....	281
inter-file inline expansion.....	40
inter-file inline expansion directory specification.....	11
inter-module optimization.....	28
internal.....	546, 815, 903
internal representation .....	288
internal representation of floating-point numbers.....	302
internal symbol .....	633
intrinsic function.....	311, 361
intrinsic functions .....	927
iomanip .....	533
ios .....	533
ios class.....	541
ios class manipulators .....	544
ios::~~ios .....	542
ios::bad.....	543
ios::clear.....	542
ios::copyfmt .....	543
ios::eof .....	543
ios::fail .....	543
ios::good .....	542
ios::init.....	542
ios::ios.....	542
ios::operator void* .....	542
ios::operator!.....	542
ios::rdbuf.....	543
ios::rdstate.....	542
ios::sb.....	541
ios::setstate.....	542
ios::state .....	541
ios::tie .....	543
ios::tiestr .....	541
ios_base class.....	536
ios_base::_ec2p_copy_base .....	539
ios_base::_ec2p_init_base .....	539
ios_base::~~ios_base .....	539

ios_base::adjustfield .....	537
ios_base::app .....	538
ios_base::ate.....	538
ios_base::badbit .....	538
ios_base::basefield .....	537
ios_base::beg .....	539
ios_base::binary .....	538
ios_base::boolalpha.....	537
ios_base::cur .....	539
ios_base::dec.....	537
ios_base::end .....	539
ios_base::eofbit .....	538
ios_base::failbit.....	538
ios_base::fill.....	540
ios_base::fillch.....	536
ios_base::fixed .....	537
ios_base::flags.....	539
ios_base::floatfield.....	537
ios_base::fmtfl .....	536
ios_base::fmtflags .....	537
ios_base::fmtmask .....	537
ios_base::goodbit .....	538
ios_base::hex .....	537
ios_base::in .....	538
ios_base::Init class .....	535
ios_base::Init::~Init.....	535
ios_base::Init::Init .....	535
ios_base::Init::init_cnt .....	535
ios_base::internal .....	537
ios_base::ios_base.....	539
ios_base::iostate.....	538
ios_base::left.....	537
ios_base::oct .....	537
ios_base::openmode.....	538
ios_base::out .....	538
ios_base::prec .....	536
ios_base::precision.....	540
ios_base::right.....	537
ios_base::scientific.....	537
ios_base::seekdir.....	539
ios_base::setf .....	539
ios_base::showbase.....	537
ios_base::showpoint.....	537

ios_base::showpos .....	537
ios_base::skipws .....	537
ios_base::statemask .....	538
ios_base::trunc .....	538
ios_base::unitbuf .....	537
ios_base::unsetf .....	539
ios_base::uppercase .....	537
ios_base::wide .....	536
ios_base::width .....	540
iostate .....	538
iostream .....	533
isalnum .....	402
isalpha .....	402, 403
iscntrl .....	402, 403
isdigit .....	404
isgraph .....	404
islower .....	402, 405
isprint .....	402, 405
ispunc .....	406
isspace .....	406
istream .....	533
istream class .....	556
istream class manipulator .....	564
istream non-member function .....	564
istream::_ec2p_getistr .....	559
istream::~istream .....	559
istream::chcount .....	556
istream::gcount .....	560
istream::get .....	560, 561
istream::getline .....	561
istream::ignore .....	562
istream::istream .....	559
istream::operator>> .....	559, 560
istream::peek .....	562
istream::putback .....	562
istream::read .....	562
istream::readsome .....	562
istream::seekg .....	563
istream::sentry class .....	555
istream::sentry::~sentry .....	555
istream::sentry::ok_ .....	555
istream::sentry::operator bool .....	555
istream::sentry::sentry .....	555

istream::sync .....	563
istream::tellg .....	563
istream::unget .....	563
isupper .....	402, 407
isxdigit .....	407
iterated expansion .....	756, 758

## **J**

Japanese code conversion in object code.....	63
jmp_buf.....	444
joining sections .....	191

## **K**

keyword .....	311
---------------	-----

## **L**

L_tmpnam.....	453
label .....	627
labs.....	510
lang .....	61
language specifications .....	279
latin1 .....	62, 94
LDBL_DIG.....	411
LDBL_EXP_DIG .....	411
LDBL_MANT_DIG .....	411
LDBL_MAX .....	409
LDBL_MAX_10_EXP .....	410
LDBL_MAX_EXP .....	410
LDBL_MIN .....	410
LDBL_MIN_10_EXP .....	411
LDBL_MIN_EXP .....	410
LDBL_NEG_EPS .....	412
LDBL_NEG_EPS_EXP .....	412
LDBL_POS_EPS.....	412
LDBL_POS_EPS_EXP .....	412
ldexp .....	424
ldexpf.....	439
ldiv .....	510
ldiv_t.....	499
left.....	60, 546
legacy.....	21
length .....	23
libraries .....	390

libraries unsupported .....	625
library.....	4, 39, 103, 109, 533
library file .....	103
library information.....	182
library listings .....	180
library types .....	390
limitations .....	921
limits.h .....	413
lines.....	97
linkage listing .....	172
linkage listings.....	171
linkage map information.....	174
linking C/C++ programs and assembly programs.....	244
list .....	22, 79, 117
list contents .....	117
list contents and format.....	23
list file .....	22, 117
list file specification.....	941
list options.....	22, 78, 117
listing .....	157, 167, 171, 180
little-endian space .....	351
local label.....	648
local symbol name hide .....	135
locale.h.....	625
LOCATE .....	681
location counter .....	631, 637
log.....	424, 587, 597
log10 .....	425, 587, 598
log10f.....	440
logf.....	439
logo .....	62, 98, 139
long .....	289
long double .....	289, 302, 305
LONG_MAX.....	413
LONG_MIN .....	413
longjmp.....	447
longreg.....	55
loop .....	28
loop expansion maximum number specification .....	36
LOW .....	644
lowercase letter .....	401
low-level interface routine .....	206, 215, 222
lseek .....	219

LWORD..... 644

## **M**

M..... 52, 90, 678  
mac ..... 372  
MAC ..... 361  
MAC register ..... 44  
machine.h..... 361  
machinecode ..... 14  
macl ..... 372  
macro body ..... 774, 780  
macro call ..... 774, 775  
macro definition..... 774, 775  
macro function..... 774  
macro function directives..... 776  
macro generation number ..... 781  
macro name..... 774  
macro name definition ..... 10  
macro replacement processing exclusion..... 783  
macsave ..... 44  
malloc ..... 506  
manipulating character arrays..... 511  
mantissa ..... 302  
map ..... 113  
math.h ..... 415  
mathematical operations ..... 415, 429  
mathf.h..... 429  
max\_unroll..... 36  
MD..... 52, 90, 678  
memchr ..... 521  
memcmp ..... 518  
memcpy ..... 514  
memmove ..... 532  
memory ..... 131  
memory allocation ..... 195  
memory area allocated for parameters ..... 251  
memory indirect addressing mode ..... 31  
memory management library ..... 576  
memory occupancy reduction ..... 131  
memset..... 530  
message..... 10, 114  
message level ..... 42, 48, 135  
method for mutual referencing of external names ..... 244



mnemonic .....	650
modf.....	425
modff .....	440
module extraction .....	134
module information within library.....	183
module replacement.....	133
movfpe.....	377
movmdb.....	381
movmdl.....	381
movmdw .....	381
movsd.....	382
movtpe .....	378
msg_unused .....	115
mulsu .....	374
multiplier .....	678
muluu.....	374
mystrbuf.....	573

## *N*

naming files .....	155
new .....	576
new_handler.....	576
no_float.h.....	498
noalign .....	19
noboolalpha .....	544
nocompress .....	131
nocpuexpand.....	16
nocross_reference .....	82
nodebug .....	14, 72, 111
noexception.....	57
noexclude.....	86
noline .....	48, 145
nolist .....	22, 79
nologo .....	62, 98, 139
nolongreg .....	55
nomessage.....	10, 114
none .....	18
noobject .....	18, 77
nooptimize .....	74, 120
nop .....	382
noprelink.....	4, 106
noregexpansion .....	46
norm.....	587, 597

normalization .....	395
normalized number .....	303, 304, 305
noscope .....	40
nosection .....	83
noshow .....	81
noshowbase .....	545
noshowpoint .....	545
noshowpos .....	545
noskipws .....	544, 545
nosource .....	80
nostructreg .....	54
not-a-number .....	303, 305, 306, 457
notification of unreferenced symbol .....	115
NOTOPN .....	414, 884
nouppercase .....	546
novolatile .....	33
NULL .....	393, 394, 397
null character .....	394

## O

object .....	18, 23, 77, 109
object file format .....	956
object file output .....	18
object information .....	164
object options .....	12, 71
object type .....	14
oct .....	546
octal number .....	635
off_type .....	534
open .....	216
operand .....	628
operation .....	627
operation size .....	650
operation size expanded interpretation .....	16
operator .....	638
operator delete .....	577
operator delete[ ] .....	577
operator evaluation order .....	310
operator new .....	577
operator new[ ] .....	576, 577
operator!= .....	586, 596, 618
operator- .....	585, 595
operator* .....	585, 595

operator/.....	585, 595
operator+.....	585, 595, 618
operator<.....	618
operator<< .....	571, 586, 596, 619
operator<= .....	619
operator= = .....	586, 596, 618
operator>.....	618
operator>= .....	619
operator>> .....	564, 586, 596, 619
opt_range .....	34
optimization .....	27
optimization for speed .....	28
optimization partially disabled.....	123
optimize .....	27, 74, 120
optimize options.....	25, 119
option consistency .....	937
option information .....	173, 181
options .....	7
options other than above.....	60, 93, 139
or_ccr.....	367
or_exr.....	371
ordinary characters.....	465, 471
ostream.....	533
ostream class.....	566
ostream class manipulator.....	570
ostream non-member function .....	571
ostream::~ostream.....	567
ostream::flush .....	569
ostream::operator<<.....	568
ostream::ostream .....	567
ostream::putc .....	568
ostream::seekp .....	569
ostream::sentry class.....	565
ostream::sentry::_ec2p_os .....	565
ostream::sentry::~sentry.....	565
ostream::sentry::ok_ .....	565
ostream::sentry::operator bool .....	565
ostream::sentry::sentry.....	565
ostream::tellp .....	569
ostream::write .....	568
other options .....	41, 86, 129
outcode.....	63, 96
output.....	112, 143

output file.....	112
output format .....	108
output of external symbol allocation information file.....	113
output options .....	107
output to unused areas .....	107, 113
overflow.....	307
overflow testing .....	361
overview .....	925
overview of formats.....	471
ovfaddc .....	383
ovfaddl.....	383
ovfadduc .....	383
ovfaddul.....	383
ovfadduw .....	383
ovfaddw .....	383
ovfnegc .....	387
ovfnegl.....	387
ovfnegw .....	387
ovfshalc.....	385
ovfshall .....	385
ovfshalw.....	385
ovfshlluc .....	386
ovfshllul.....	386
ovfshlluw .....	386
ovfsubc.....	384
ovfsubl.....	384
ovfsubuc.....	384
ovfsubul .....	384
ovfsubuw .....	384
ovfsubw .....	384

## **P**

P.....	186, 198, 315
pack.....	58
parameter allocation.....	253
parameter assignment examples .....	257
parameter size specification.....	468
parameter storage register .....	53
parameters and return values .....	249
passing parameters.....	249
path .....	149
perror .....	497
pointer.....	283, 289

pointer size.....	321
pointer size specification .....	32
pointer to data member .....	290
pointer to function member.....	291
pointer to virtual function member .....	291
pointers to function members .....	292
polar .....	587, 597
pos_type.....	534
pow .....	426, 588, 598
PowerON_Reset .....	205, 208
powf .....	441
precision.....	467
preferential allocation of register storage class variables .....	49
preinclude .....	9
prelinker.....	106
preprocessor.....	13
preprocessor expansion.....	13, 48
preprocessor specifications.....	287
preprocessor variable reference in macro .....	781
preprocessor variables.....	752
printf .....	456, 474
printing character .....	401
procedures for developing programs .....	1
profile.....	122
profile information.....	122
program.....	15
program area.....	198
program development .....	276
ptr16.....	32
ptr16 option.....	291
ptrdiff_t.....	397
PTRERR .....	414, 883
putc .....	488
putchar .....	489
puts .....	489

## **Q**

qsort .....	508
qualifier specifications.....	285

## **R**

radix .....	395
rand .....	504

RAND_MAX.....	499
range error.....	416, 430
range of integer types and values.....	281
read .....	218
real .....	586, 596
realloc .....	506
record.....	111
record size unification.....	111
reduce empty areas of boundary alignment .....	115
reent .....	144
reentrant library .....	237, 620
reference .....	290
regexpansion.....	46
register .....	28, 120, 284
register allocation of 4-byte parameters.....	55
register allocation of structure parameters.....	54
register specifications .....	284
register usage .....	267
regparam .....	53
relative .....	185
relocate.....	109
rename .....	132
replace.....	133
replacement symbols .....	753
reserved word.....	631, 633
resetiosflags .....	572
return code .....	394
return value setting .....	254
rewind .....	495
right.....	60, 546
rom.....	16, 112
ROM support function.....	112
ROM, RAM allocation .....	198
rotlc.....	375
rotll .....	375
rotlw.....	375
rotrc.....	375
rotrl .....	375
rotrw .....	375
rounding.....	307, 395
rtti .....	57
rules concerning allocation and release of stack frames .....	246
rules concerning registers .....	247

rules concerning the stack pointer.....	246
runtime type information .....	57

## S

S .....	188
S9 .....	130
safe .....	121
same_code .....	120
samecode_forbid .....	123
samesize .....	121
SBR .....	59, 91, 138, 339, 679
SBR address specification .....	138
sbrk .....	220
scalar type .....	288
scanf .....	475
SCHAR_MAX .....	413
SCHAR_MIN .....	413
scientific .....	546
scope .....	40
sdebug .....	<i>See</i>
section .....	15, 83
section address .....	125
section address operator .....	311, 359
section attribute .....	185
section information listing .....	171
section information within library .....	183
section initialization .....	206
section initialization tables .....	205
section name .....	631
section options .....	125
sections .....	185
SEEK_CUR .....	453, 493
SEEK_END .....	453, 493
SEEK_SET .....	453, 493
selecting C or C++ language .....	61
set_ccr .....	366
set_exr .....	369
set_imask_ccr .....	365
set_imask_exr .....	368
set_new_handler .....	577
set_vbr .....	372
setbase .....	572
setbuf .....	463

setfill .....	572
setiosflags .....	572
setjmp.....	446
setjmp.h.....	444
setprecision .....	572
settings for the program execution environment.....	195
setvbuf .....	464
setw .....	572
shift .....	28
short .....	289
short absolute addressing mode .....	32
short_format.....	120, 121
show .....	23, 81, 117
showbase.....	545
showpoint.....	545
showpos .....	545
SHRT_MAX.....	413
SHRT_MIN .....	413
sign .....	302
sign extension .....	299
signal.h.....	625
signal_sem .....	215, 221, 237
signed char .....	280, 289
sin .....	420, 588, 598
sinf .....	435
single-precision.....	429
sinh .....	422, 588, 598
sinhf .....	437
size .....	288, 291
size_t.....	397
SIZEOF.....	642
sjis.....	62, 63, 95
skipws .....	545
sleep .....	376
smanip class manipulator.....	572
source.....	23, 80
source listing .....	158, 167
source options .....	8, 66
source statements .....	627
space .....	102, 113, 121
space characters .....	456
special character.....	401
speed .....	28, 121



sprintf.....	476
sqrt .....	426, 588, 598
sqrtf .....	441
srand .....	504
sring literal.....	647
sscanf .....	477
subcommand file option.....	137
stack .....	56, 130, 185
STACK .....	681
stack analysis tool.....	147
stack area .....	199
stack area usage .....	267
stack information file.....	130
stack section creation.....	314
stack size specification.....	56
standard error output file.....	394
standard include file.....	390
standard input file .....	394
standard input/output files .....	394
standard output file .....	394
start .....	19, 31, 32, 125
STARTOF .....	642
statement specifications .....	286
static .....	18
static memory area allocation .....	196
statistics .....	23
statistics information.....	166
stdarg.h .....	448
stddef.h .....	397
stderr .....	394, 453
stdin .....	394, 453
stdio.h .....	453
stdlib.h .....	499
stdout .....	394, 453
strcat .....	516
strchr .....	522
strcmp .....	519
strcpy .....	514
strcspn .....	523
stream input/output .....	392
stream input/output class library .....	533
streambuf .....	533
streambuf class.....	547

streambuf::_B_cnt_ptr .....	547
streambuf::_B_len_ptr .....	547
streambuf::~streambuf .....	550
streambuf::B_beg_pptr .....	547
streambuf::B_beg_ptr .....	547
streambuf::B_end_ptr .....	547
streambuf::B_next_pptr .....	547
streambuf::B_next_ptr .....	547
streambuf::C_flg_ptr .....	547
streambuf::eback .....	552
streambuf::egptr .....	552
streambuf::epptr .....	553
streambuf::gbump .....	552
streambuf::gptr .....	552
streambuf::in_avail .....	550
streambuf::overflow .....	554
streambuf::pbackfail .....	554
streambuf::pbase .....	552
streambuf::pbump .....	553
streambuf::pptr .....	552
streambuf::pubseekoff .....	550
streambuf::pubseekpos .....	550
streambuf::pubsetbuf .....	550
streambuf::pubsync .....	550
streambuf::sbumpc .....	551
streambuf::seekoff .....	553
streambuf::seekpos .....	553
streambuf::setbuf .....	553
streambuf::setg .....	552
streambuf::setp .....	553
streambuf::sgetc .....	551
streambuf::sgetn .....	551
streambuf::showmanyc .....	553
streambuf::snextc .....	551
streambuf::sputbackc .....	551
streambuf::sputc .....	551
streambuf::sputn .....	552
streambuf::streambuf .....	550
streambuf::sungetc .....	551
streambuf::sync .....	553
streambuf::uflow .....	554
streambuf::underflow .....	554
streambuf::xsgetn .....	554

streambuf::xsputn .....	554
streamoff .....	534
streamsize .....	534
strerror .....	531
strict_ansi .....	49
string .....	15, 599
string class .....	599
string class manipulator .....	616
string handling class library .....	599
string literal manipulation functions in macro .....	784, 786
string literal output area .....	15
string.h .....	511
string::~string .....	606
string::append .....	609
string::assign .....	609, 610
string::at .....	608
string::begin .....	607
string::c_str .....	612
string::capacity .....	608
string::clear .....	608
string::compare .....	615
string::const_iterator .....	599
string::copy .....	612
string::data .....	612
string::empty .....	608
string::end .....	607
string::erase .....	610, 611
string::find .....	612
string::find_first_not_of .....	614
string::find_first_of .....	613
string::find_last_not_of .....	614
string::find_last_of .....	613
string::insert .....	610
string::iterator .....	599
string::length .....	607
string::max_size .....	607
string::npos .....	599
string::operator[ ] .....	608
string::operator+= .....	608, 609
string::operator= .....	606, 607
string::replace .....	611, 612
string::reserve .....	608
string::resize .....	607, 608

string::rfind .....	612, 613
string::s_len .....	599
string::s_ptr .....	599
string::s_res .....	599
string::size .....	607
string::string .....	606
string::substr .....	614
string::swap .....	612
string_unify .....	120
strip .....	134
strlen .....	531
strncat .....	517
strncmp .....	520
strncpy .....	515
strpbrk .....	524
strchr .....	525
strspn .....	526
strstr .....	527
strtod .....	500, 502
strtok .....	528
strtol .....	503
struct .....	28
struct_alloc .....	38
structreg .....	54
structure .....	285
structure data allocation .....	294
structure type .....	293
structure/union member register allocation .....	38
structured .....	81
structured assembly .....	790
structured assembly directives .....	792
structured assembly symbol .....	633
stype .....	109
subcommand .....	63, 99, 137
subcommand file .....	63, 137
swap .....	619
switch .....	28
switch statement output code selection method .....	29
symbol address file .....	126
symbol allocation information .....	161
symbol definition .....	104
symbol information .....	174
symbol information within library .....	183

symbol name deletion .....	132
symbol name modification.....	132
symbol_delete .....	120
symbol_forbid.....	123
symbols .....	631
SYS_OPEN .....	453
sysrof .....	959

## ***T***

tab .....	23
table .....	29
tables for section initialization.....	210
tan .....	421, 588, 598
tanf .....	436
tanh .....	422, 588, 598
tanhf .....	437
tas.....	378
template .....	18
template instance generation.....	18
term.....	638
termination processing.....	140
termination processing routine.....	206
termination processing routines .....	240
terms used in library function descriptions .....	392
text file.....	394
time.h .....	625
TMP_MAX.....	453
tolower.....	408
toupper .....	408
trapa .....	376
tuning options .....	84
type conversion.....	249

## ***U***

UCHAR_MAX .....	413
UINT_MAX .....	413
ULONG_MAX .....	413
underflow .....	307
ungetc.....	490
uninitialized data areas.....	198
uninitialized data section address area .....	198
union .....	285
union data allocation.....	295

union type .....	293
unsigned char .....	289
unsigned int.....	289
unsigned long.....	289
unsigned short.....	289
uppercase .....	545
uppercase letter .....	401
use of EEPMOV/P.W .....	925, 930
used.....	18
USHRT_MAX.....	413

## V

va_arg .....	451
va_end.....	452
va_list.....	448, 451
va_start.....	450, 451
vacant loop elimination.....	36
variable access optimization symbol information.....	176
variable_access .....	120
variable_forbid.....	123
VEC_TBL.....	205, 207
vector table .....	205
vector table settings .....	207
Ver.4.0 Optimization Object Code .....	21
verify options .....	127
version upgrade.....	933
vfprintf.....	478
virtual function table.....	296
virtual function table area .....	198
volatile .....	33
volatile_loop .....	45
vprintf .....	479
vsprintf.....	480

## W

wait_sem .....	215, 220, 237
warning .....	44, 48, 135, 815, 885, 903, 917
white-space character.....	401
white-space characters .....	471
width .....	23
write .....	219
write operation.....	185
ws.....	564

## ***X***

xor_ccr.....	368
xor_exr.....	371

## ***Z***

zero .....	303, 305, 306
------------	---------------





---

**Renesas Microcomputer Development Environment System  
User's Manual  
H8S, H8/300 Series C/C++ Compiler, Assembler,  
Optimizing Linkage Editor**

Publication Date: Rev.1.00, January 12, 2005

Published by: Sales Strategic Planning Div.  
Renesas Technology Corp.

Edited by: Technical Documentation & Information Department  
Renesas Kodaïra Semiconductor Co., Ltd.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

---



## RENESAS SALES OFFICES

<http://www.renesas.com>

Refer to "<http://www.renesas.com/en/network>" for the latest and detailed information.

### **Renesas Technology America, Inc.**

450 Holger Way, San Jose, CA 95134-1368, U.S.A  
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

### **Renesas Technology Europe Limited**

Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K.  
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

### **Renesas Technology Hong Kong Ltd.**

7th Floor, North Tower, World Finance Centre, Harbour City, 1 Canton Road, Tsimshatsui, Kowloon, Hong Kong  
Tel: <852> 2265-6688, Fax: <852> 2730-6071

### **Renesas Technology Taiwan Co., Ltd.**

10th Floor, No.99, Fushing North Road, Taipei, Taiwan  
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

### **Renesas Technology (Shanghai) Co., Ltd.**

Unit2607 Ruijing Building, No.205 Maoming Road (S), Shanghai 200020, China  
Tel: <86> (21) 6472-1001, Fax: <86> (21) 6415-2952

### **Renesas Technology Singapore Pte. Ltd.**

1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632  
Tel: <65> 6213-0200, Fax: <65> 6278-8001

H8S, H8/300 Series  
C/C++ Compiler, Assembler,  
Optimizing Linkage Editor  
User's Manual



Renesas Technology Corp.

2-6-2, Ote-machi, Chiyoda-ku, Tokyo, 100-0004, Japan